Runtime Verification of Component-Based Systems in the BIP Framework with Formally-Proved Sound and Complete Instrumentation

Yliès Falcone[1], Mohamad Jaber[2], Thanh-Hung Nguyen[3], Marius Bozga[4], Saddek Bensalem[4]
Ylies.Falcone@ujf-grenoble.fr[1], mj54@aub.edu.lb[2],
hungnt@soict.hut.edu.vn[3], and FirstName.LastName@imag.fr[4]

[1] Laboratoire d'Informatique de Grenoble, UJF - University of Grenoble I, Grenoble, France
[2] American University of Beirut, Beirut, Lebanon
[3] Hanoi University of Sciences and Technology, Hanoi, Vietnam
[4] VERIMAG Laboratory, UJF - University of Grenoble I, Grenoble, France

Abstract. Verification of *component-based systems* still suffers from limitations such as state space explosion since a large number of different components may interact in an heterogeneous environment. These limitations entail the need for complementary verification methods such as *runtime verification* based on dynamic analysis and apt to scalability.

In this paper, we integrate runtime verification into the BIP (Behavior, Interaction and Priority) framework. BIP is a powerful and expressive component-based framework for the formal construction of heterogeneous systems. Our method augments BIP systems with monitors to check specifications at runtime. This method has been implemented in RV-BIP, a prototype tool that we used to validate the whole approach on a robotic application.

1 Introduction

Component-based Systems. A component-based approach consists in building complex systems by composing components (building blocks). This confers numerous advantages (e.g., productivity, incremental construction, compositionality) that allow one to deal with complexity in the construction phase. Component-based systems (CBS) are desirable because they allow reuse of sub-systems as well as their incremental modification without requiring global changes. The development of CBS requires methods and tools supporting a concept of architecture which characterizes the coordination between components. An architecture structures a system and involves components and relationships between the externally visible properties of those components. The global behavior of a system can, in principle, be inferred from the behavior of its components and its architecture. Component-based design is based on the separation between coordination and computation. Systems are built from units executing sequential code insulated from concurrent execution issues. The isolation of coordination mechanisms allows a global treatment and analysis on coordination constraints between components even if local computations on components are not visible (i.e., components are "black boxes").

BIP (Behavior Interaction Priority). BIP is a general framework supporting rigorous design. BIP uses a dedicated language and an associated toolset supporting the design flow. The BIP language allows building complex systems by coordinating the behavior of a set of atomic components. Behavior is described with

Labelled Transition Systems (LTS) extended with data and functions written in C. The description of coordination between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities between the interactions and is used also to express scheduling policies. The combination of interactions and priorities characterizes the overall architecture of a system and confers strong expressiveness that cannot be matched by other existing formalism dedicated to CBS [**?**]. Moreover, BIP has a rigorous operational semantics: the behavior of a composite component is formally described as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model and its implementation.

Runtime verification (monitoring). Runtime Verification (RV) [**?**,**?**,**?**] is an effective technique to ensure, at runtime, that a system respects or meets a desirable behavior. It can be used in numerous application domains, and more particularly when integrating together unreliable software components. In RV, a run of the system under scrutiny is analyzed incrementally using a decision procedure: a *monitor*. This monitor may be generated from a user-provided high level specification (e.g., a temporal formula, an automaton). This monitor aims to detect violation or satisfaction w.r.t. the given specification. Generally, it is a state machine processing an execution sequence (step by step) of the monitored program, and producing a sequence of verdicts (truth-values taken from a truth-domain) indicating specification fulfillment or violation. Recently [**?**] a new framework has been introduced for runtime verification. This expressive framework, leveraged by a finite-trace semantics (initially proposed in [**?**]) and an expressive truth-domain, allows to monitor all specifications expressing a linear temporal behavior. For a monitor to be able to observe runs of the system, the system should be instrumented in such a way that at runtime, the program sends relevant events that are consumed by the monitor. Usually, one of the main challenges when designing an RV framework is its performance. That is, adding a monitor in the system should not deteriorate executions of the initial system time and memory wise.

Motivations for using monitoring to validate component-based systems. As is the case with monolithic systems, monitoring is a complementary technique to validate the behavior of component-based systems. Monitoring has several advantages when compared to static validation techniques. Compared to static analysis, monitoring allows to check more expressive behavioral specifications. Moreover, monitoring does not rely on abstracting or over-approximating the state-space, and thus does not produce false positives. Compared to model-checking, monitoring is less sensitive to the state-explosion problem which is rapidly occurring when composing the behavior of several components. Compared to compositional verification techniques, monitoring remains applicable for BIP component-based systems (where external functions can be called). Regarding BIP systems, classical model-checking techniques rapidly become unusable because of the state-explosion problem. Consequently, the currently available verification techniques are based on compositional/incremental verification. However, in the BIP framework, compositional verification is currently limited to the verification of safety properties, and, more expressive properties such as liveness properties remain out of the scope. In Section **??**, we provide a more detailed comparison with static-verification techniques.

Challenges in monitoring component-based systems and BIP systems. Contrarily to monolithic systems (written for instance in Java), component-based systems are not endowed with an *automatic and effective instrumentation technique*. An automatic instrumentation technique allows the programmer to only indicate points of observation (in a more or less abstract fashion) and automatically adds observation code at relevant places in the original program. An effective instrumentation techniques ensures that the performance of the instrumented program is close to the performance of the original program. Such an instrumentation technique is required when designing or implementing a runtime verification framework for a target system-domain. Consequently, we should design an instrumentation technique that should enjoy several features. As is the case with monolithic system, we require an automatic and efficient instrumentation technique. Moreover, we require a high-level of confidence on its correctness. The instrumentation technique should not alter the behavior of the initial system. Existing RV frameworks attempt to preserve the behavior of the system by performing only observations. For instance, using aspect-oriented programming, the used aspects only pick-up events without modifying the control-flow of the original program. However, in the context of component-based systems, such a desirable correctness property is harder to obtain. Indeed, to monitor a component-based system, one needs to add the monitor as a new component, which is allowed to observe the system by adding interactions. Such interactions should be inserted carefully because they could not only modify the existing interactions but also modify the internal behaviors of existing components. Finally, we aim at providing a method that can be used in critical systems and thus require a high-level of confidence on the proposed instrumentation technique. Consequently, the designed instrumentation technique should be defined formally and its correctness formally proved.

Our approach (informal view). Let us depict a high-level view of our approach. On the one hand, we consider an initial component-based system $C = (B_1, \ldots, B_n)$ built over existing atomic components $B_1, \ldots, B_n$. The atomic components are designed independently regarding their data and their behavior but they can interact and synchronize at runtime. We assume a global clock operating on C in such a way that, at each execution step, each atomic component can perform one action independently. When components are interacting together, we assume this communication to be reliable and do not consider any issue regarding privacy or security. On the other hand, a property φ specifies the desired runtime behavior of C. The property φ is associated to an abstract decision procedure (a monitor) M_φ which is generated using standard monitor synthesis techniques. Verdicts over the behavior are based on an expressive truth-domain as in [**?**]. Verdicts can indicate current satisfaction and violation, or, definitive satisfaction or violation (in which case no monitoring is needed).

We shall transform C into a new component-based system $C' = (B'_1, \ldots, B'_n, M'_\varphi)$. C contains a monitor M'_φ as a component generated from M_φ. Atomic components are instrumented (B_i transformed to B'_i) to interact with M'_φ. Moreover, we require some behavioral equivalence between the initial and transformed systems. This behavioral equivalence states that, up to some behavioral equivalence relation, not only the sets of possible behaviors in the initial and transformed system are equivalent, but, in addition, if the initial system produces any execution, then the transformed system produces an equivalent execution. At runtime, the component monitor M'_φ observes the relevant pieces of information in $B'_1, \ldots, B'_n$ and emits verdicts

according to the satisfaction or violation of φ. The amount of information observed in the components $B'_1, \ldots, B'_n$ is kept minimal so as to minimize the overhead induced by the monitoring process.

Contributions. We introduce a complementary validation technique for CBS in general and BIP systems in particular. More precisely, the contributions of this paper are as follows:

1. To propose a minimal formal introduction to BIP systems providing a framework for rigorous design of CBS.
2. To extend the BIP framework by integrating an expressive RV framework previously introduced in [**?**]. We extend this framework for CBS by proposing a formal instrumentation technique that allows to observe the relevant parts in the behavior of a BIP system. This instrumentation technique is formally defined, proved sound and complete, and leverages the formal semantics of BIP.
   Given a specification, our method uniformly integrates a monitor as an additional component in a BIP system that is able to runtime check the satisfaction or violation of the specification.
3. To propose an implementation of the RV framework, RV-BIP, allowing to automatically instrument BIP systems from the description of an abstract monitor. Thanks to the code generator of BIP, the generated self-monitoring system can be directly translated into an actual C module embedded in the global system whose behavior is checked at runtime against the specification.
4. To evaluate and validate the relevance of the whole approach on a real-world application.

This paper extends a previous contribution [**?**] that appeared in the 9^{th} International Conference on Software Engineering and Formal Methods, with the following additional contributions:

- to propose a more complete and improved introduction to the BIP framework;
- to propose a detailed and rigorous proof of the correctness of the approach proposed in this paper;
- to improve the presentation and editorial quality of the previous paper by (i) formalizing some concepts that remained informal in the conference version, (ii) providing more detailed explanations in each section, (iii) correcting typos, (iv) and illustrating the concepts with additional examples;
- to propose additional experiments on our case study;
- to propose a deeper study of related work.

Paper Organization. The paper is structured as follows. Section **??** introduces the preliminary concepts needed in this paper. In Section **??** we give a minimal introduction to the BIP framework. Section **??** defines an abstract RV framework for CBS described in BIP. Section **??** shows how the abstract RV framework is implemented for BIP systems. Section **??** describes RV-BIP, an implementation used to evaluate our method on a robot application. Section **??** is dedicated to related work. Section **??** raises some concluding remarks and open perspectives. Finally, to keep our RV framework for CBS intuitive, some proofs and proof-specific definitions are omitted in Section **??**, complete proofs are given in Appendix **??**.

2 Preliminaries and Notations

We introduce some preliminary concepts and notation.

Functions and partial functions. For two domains of elements E and F, we note $[E \to F]$ (resp. $[E \rightharpoonup F]$) the set of functions (resp. partial functions) from E to F. When elements of E depend on the elements of F, we note $\{e \in E\}_{f \in F'}$, where $F' \subseteq F$, for $\{e \in E \mid f \in F'\}$ or $\{e\}_{f \in F'}$ when clear from context. For two functions $v \in [X \to Y]$ and $v' \in [X' \to Y']$, the substitution function noted v/v', where $v/v' \in [X \cup X' \to Y \cup Y']$, is defined as follows:

$$v/v'(x) = \begin{cases} v'(x) \text{ if } x \in X', \\ v(x) \text{ otherwise.} \end{cases}$$

A predicate over some domain E is a function in the set $[E \to \{\texttt{true}, \texttt{false}\}]$ where `true` and `false` are the usual Boolean constants. Given, some predicate p over some domain E and some element $e \in E$, we abbreviate $p(e) = \texttt{true}$ (resp. $p(e) = \texttt{false}$) by $p(e)$ (resp. $\neg p(e)$). Given some sets of functions $[X_1 \to Y_1], \ldots, [X_n \to Y_n]$, the set of consistent merges of these functions, denoted $\biguplus \{[X_1 \to Y_1], \ldots, [X_n \to Y_n]\}$ is defined as:

$$\biguplus \{[X_1 \to Y_1], \ldots, [X_n \to Y_n]\} \stackrel{\text{def}}{=} \Big\{ f \in [\bigcup_{i=1}^{n} X_i \to \bigcup_{i=1}^{n} Y_i] \mid \forall i \in [1, n] : x \in X_i \Rightarrow f(x) \in Y_i \Big\}.$$

That is, the set of consistent merges is the set of functions from the union of domains ($\bigcup_{i=1}^{n} X_i$) to the union of co-domains ($\bigcup_{i=1}^{n} Y_i$) where only consistent data bindings are allowed.

Pattern-matching. We shall use the mechanism of pattern-matching to define some functions more concisely. We recall an intuitive definition for the sake of completeness. Evaluating the expression:

```
match expression with
    | pattern_1 → expression_1
    | pattern_2 → expression_2
    ...
    | pattern_n → expression_n
```

consists in comparing successively `expression` with the patterns `pattern_1`, ..., `pattern_n` in order. When a pattern `pattern_i` fits `expression`, then the associated `expression_i` is returned.

Sequences. Given a set of elements E, a sequence or a list of length n over E is denoted $e_1 \cdot e_2 \cdots e_n$ where $\forall i \in [1, n] : e_i \in E$. When elements of a sequence are assignments, the sequence is delimited by square brackets e.g., $[x_1 := \mathit{expr}_1; \ldots; x_n := \mathit{expr}_n]$. Concatenation of assignments or sequences of assignments is denoted by ";". The empty sequence of assignments is noted $[\,]$. Assignments in a sequence are executed according to their order in the list (beginning with the first elements).[5] The set of all sequences over E is noted E^*. We shall use regular expressions to concisely denote sets of sequences over E. A regular expression over E defines a set of sequences over E. Given two regular expressions re_1, re_2 over E, $\mathit{re}_1 \cdot \mathit{re}_2$ (resp. $\mathit{re}_1 + \mathit{re}_2$) is a regular expression and denotes the set $\{s_1 \cdot s_2 \mid s_1 \in \mathit{re}_1 \wedge s_2 \in \mathit{re}_2\}$ (resp. $\mathit{re}_1 \cup \mathit{re}_2$).

[5] Consequently, it does not forbid to have several assignments to a variable in such sequences. In such a case, the last assignment to this variable determines the final value of the variable.

Transition Systems. In the following, Labelled Transition System (LTS) are used to define the semantics of BIP systems. LTSs defined over an alphabet Σ is a 3-tuple $(\mathrm{Lab}, \mathrm{Loc}, \mathrm{Trans})$ where Lab is a set of labels, Loc is a non-empty set of locations. $\mathrm{Trans} \subseteq \mathrm{Loc} \times \mathrm{Lab} \times \mathrm{Loc}$ is the transition relation. A transition $(l, e, l') \in \mathrm{Trans}$ means that the LTS can move from location l to location l' by consuming label e. We abbreviate $(l, e, l') \in \mathrm{Trans}$ by $l \xrightarrow{e}_{\mathrm{Trans}} l'$ or by $l \xrightarrow{e} l'$ when clear from context. Moreover, $l \xrightarrow{e}$ is a short for $\exists l' \in \mathrm{Loc} : l \xrightarrow{s} l'$.

Monolithic/Component-based vs Centralised/Distributed Systems. As a last preliminary notion, we stress the importance of the difference between two classifications of systems. First, systems are categorized as monolithic (vs component-based) when they are designed as a single entity (resp. an association of several entities). Second, systems are categorized as centralised (vs distributed) when they execute on a single computation unit (vs multiple computation units). Consequently, we should stress that this paper does not target runtime verification of distributed systems (cf. [**?**,**?**]). Note also, that from a component-based system, one can generate a distributed implementation (cf. [**?**]) or a centralised implementation (cf. [**?**]) by first transforming the component-based system into an equivalent monolithic system.

3 BIP - Behavior Interaction Priority

In this section we recall the necessary concepts of the BIP framework [**?**]. BIP is a component-based framework for constructing systems by superposing three layers of modeling: Behavior, Interaction, and Priority. The *behavior* layer consists of a set of atomic components represented by transition systems. The *interaction* layer models the collaboration between components. Interactions are described using sets of ports and connectors between them [**?**]. The *priority* layer is used to enforce scheduling policies applied to the interaction layer, given by a strict partial order on interactions.

3.1 Component-based Construction

BIP offers primitives and constructs for modeling and composing complex behaviors from atomic components. Atomic components are Labeled Transition Systems (LTS) extended with C functions and data. Transitions are labeled with sets of communication ports. Composite components are obtained from atomic components by specifying connectors and priorities.

Atomic Components. An atomic component is endowed with a finite set of local variables X taking values in a domain Data. Atomic components synchronize and exchange data with other components through the notion of *port*.

Definition 1 (Port). *A port $p[x_p]$, where $x_p \subseteq X$, is defined by a port identifier p and some data variables in a set x_p (referred as the support set).*

Definition 2 (Atomic component). *An atomic component B is defined as a tuple $(P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$, where:*

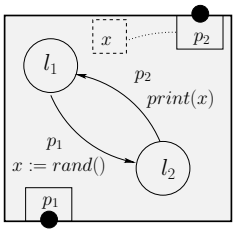


Fig. 1. An atomic component in BIP

- *(P, L, T) is an LTS over a set of ports P. L is a set of control locations and $T \subseteq L \times P \times L$ is a set of transitions.*
- *X is a set of variables.*
- *For each transition $\tau \in T$:*
  - *g_τ is a Boolean condition over X: the guard of τ,*
  - *$f_\tau \in \{x := f^x(X) \mid x \in X\}^*$: the computation step of τ, a list of statements.*

For $\tau = (l, p, l') \in T$ a transition of the internal LTS, l (resp. l') is referred as the source (resp. destination) location and p is a port through which an interaction with another component can take place. Moreover, a transition $\tau = (l, p, l') \in T$ in the internal LTS involves a transition in the atomic component of the form $(l, p, g_\tau, f_\tau, l')$ which can be executed only if the guard g_τ evaluates to `true`, and f_τ is a computation step: a set of assignments to local variables in X.

In the rest of this article, we use the dot notation to denote the elements of atomic components, e.g., $B.P$ denotes the set of ports of the atomic component B, $B.L$ denotes its set of locations, etc.

Example 1 (Atomic component). Figure **??** shows an example of an atomic component with a variable x, two ports p_1 and p_2 with support set $\{x\}$, and two control locations l_1, l_2. At location l_1, the transition labelled by port p_1 is possible (the guard evaluates to `true` by default). When an interaction through p_1 takes place, a random value is assigned to the variable x through the assignment $x := rand()$. From the control location l_2, the transition labelled by the port p_2 is possible, the variable x is not modified, and the value of x is printed and exported through p_2.

Semantics of Atomic Components. The semantics of an atomic component is an LTS over configurations and ports, formally defined as follows:

Definition 3 (Semantics of Atomic Components). *The semantics of the atomic component $(P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ is an LTS (P, Q, T_0) such that*

- $Q = L \times [X \to \text{Data}] \times (P \cup \{\texttt{null}\})$,
- $T_0 = \{((l', v', p'), p(v_p), (l, v, p)) \in Q \times P \times Q \mid \exists \tau = (l', p[x_p], l) \in T : g_\tau(v') \wedge v = f_\tau(v'/v_p)\}$, *where* $v_p \in [x_p \to \text{Data}]$.

A configuration is a triple $(l, v, p) \in Q$ where $l \in L$ is a control location, $v \in [X \to \text{Data}]$ is a valuation of the variables in X, and $p \in P$ is the port labelling the last-executed transition or `null` when no transition

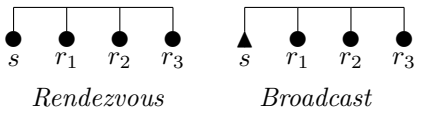


Fig. 2. Connectors in BIP

has been executed. The evolution of configurations $(l', v', p') \stackrel{p(v_p)}{\rightarrow} (l, v, p)$, where v_p is a valuation of the variables x_p attached to the port p, is possible if there exists a transition $(l', p[x_p], g_\tau, f_\tau, l)$, such that $g_\tau(v') = \texttt{true}$. As a result, the valuation v' of variables is modified to $v = f_\tau(v'/v_p)$.

Creating composite components. Assuming some available atomic components $B_1, \ldots, B_n$, we show how to connect the components in the set $\{B_i\}_{i\in I}$ with $I \subseteq [1, n]$ using a *connector*.

A connector γ is used to specify possible interactions, i.e., the sets of ports that have to be jointly executed. Two types of ports (*synchron*, *trigger*) are defined, in order to specify the feasible interactions of a connector. A *trigger* port is active: the port can initiate an interaction without synchronizing with other ports. A trigger port is graphically represented by a triangle. A *synchron* port is passive: the port needs synchronization with other ports for initiating an interaction. A synchron port is graphically represented by a circle. A feasible interaction of a connector is a subset of its ports such that either it contains some trigger, or it is maximal. Figure **??** shows two connectors: *Rendezvous* (only the maximal interaction $\{s, r_1, r_2, r_3, r_4\}$ is possible), *Broadcast* (all the interactions containing the trigger port s are possible). Formally, a connector is defined as follows:

Definition 4 (Connector). *A connector γ is a tuple $(\mathcal{P}_\gamma, t, G, F)$, where:*

- $\mathcal{P}_\gamma = \{p_i[x_i] \mid p_i \in B_i.P\}_{i\in I}$ *such that* $\forall i \in I : \mathcal{P}_\gamma \cap B_i.P = \{p_i\}$*;*
- $t \in [\mathcal{P}_\gamma \rightarrow \{\texttt{true}, \texttt{false}\}]$ *such that* $t(p) = \texttt{true}$ *if p is trigger (and* $t(p) = \texttt{false}$ *if p is synchron);*
- *G is a Boolean expression over the set of variables $\cup_{i\in I}\, x_i$ (the guard);*
- *F is an update function defined over the set of variables $\cup_{i\in I}\, x_i$.*

$\mathcal{P}_\gamma$ is the set of connected ports called the support set of γ. The ports in $\mathcal{P}_\gamma$ are tagged using function t indicating whether they are trigger or synchron. Moreover, for each $i \in I$, x_i is a set of variables associated to the port p_i.

A communication between the atomic components of $\{B_i\}_{i\in I}$ through a connector $(\mathcal{P}_\gamma, t, G, F)$ is defined using the notion of *interaction*:

Definition 5 (Interaction). *A set of ports $a = \{p_j\}_{j\in J} \subseteq \mathcal{P}_\gamma$ for some $J \subseteq I$ is an interaction of γ if one of the following conditions holds:*

1. *there exists $j \in J$ such that p_j is trigger;*
2. *for all $j \in J$, p_j is synchron and $\{p_j\}_{j\in J} = \mathcal{P}_\gamma$.*

An interaction a has a guard and two functions G_a, F_a, respectively obtained by projecting G and F on the variables of the ports involved in a. We denote by $\mathcal{I}(\gamma)$ the set of interactions of γ. Synchronization through an interaction involves two steps. First, the guard G_a is evaluated, then the update function F_a is applied.

Definition 6 (Composite Component). *A composite component is defined from a set of available atomic components $\{B_i\}_{i\in I}$ and a set of connectors Γ. The connection of the components in $\{B_i\}_{i\in I}$ using the set Γ of connectors is denoted $\Gamma(\{B_i\}_{i\in I})$.*

Note that a composite component obtained by composition of a set of atomic components can be composed with other components in a hierarchical and incremental fashion using the same operational semantics.

Definition 7 (Semantics of Composite Components). *A state q of a composite component $\Gamma(\{B_1,\ldots,B_n\})$, where Γ connects the B_i's for $i \in [1,n]$, is an n-tuple $q = (q_1,\ldots,q_n)$ where $q_i = (l_i, v_i)$ is a state of B_i. Thus, the semantics of $\Gamma(\{B_1,\ldots,B_n\})$ is precisely defined as a transition system $(Q, A, \longrightarrow)$, where:*

- *$Q = B_1.Q \times \ldots \times B_n.Q$,*
- *$A = \cup_{\gamma\in\Gamma}\{a \in \mathcal{I}(\gamma)\}$ is the set of all possible interactions,*
- *$\longrightarrow$ is the least set of transitions satisfying the following rule:*

$$\frac{\begin{array}{cc}\exists\gamma\in\Gamma : \gamma = (P_\gamma, t, G, F) & \exists a \in \mathcal{I}(\gamma) \qquad G_a(v(X)) \\ \forall i \in I :\ q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = F_{a_i}(v(X)) & \forall i \notin I :\ q_i = q_i'\end{array}}{(q_1,\ldots,q_n) \xrightarrow{a} (q_1',\ldots,q_n')}$$

where $a = \{p_i\}_{i\in I}$, X is the set of variables attached to the ports of a, v is the global valuation of variables, and F_{a_i} is the partial function derived from F restricted to the variable associated to p_i.

The meaning of the above rule is the following: if there exists an interaction a such that all its ports are enabled in the current state and its guard ($G_a(v(X))$) evaluates to `true`, then the interaction can be fired. When a is fired, not-involved components remain in the same state and involved components evolve according to the interaction.

Notice that several distinct interactions can be enabled at the same time, thus introducing non-determinism in the product behavior. One can add priorities to reduce non-determinism. In this case, one of the interactions with the highest priority is chosen non-deterministically.[6]

Definition 8 (Priority). *Let $C = (Q, A, \longrightarrow)$ be the behavior of the composite component $\Gamma(\{B_1,\ldots,B_n\})$. A* priority model *$\pi$ is a strict partial order on the set of interactions A. Given a priority model π, we abbreviate $(a, a') \in \pi$ by $a \prec_\pi a'$ or $a \prec a'$ when clear from the context. Adding the priority model π over $\Gamma(\{B_1,\ldots,B_n\})$ defines a new composite component $\pi\big(\Gamma(\{B_1,\ldots,B_n\})\big)$ noted $\pi(C)$ and whose behavior is defined by $(Q, A, \longrightarrow_\pi)$, where $\longrightarrow_\pi$ is the least set of transitions satisfying the following rule:*

$$\frac{q \xrightarrow{a} q' \qquad \neg\big(\exists a' \in A, \exists q'' \in Q : a \prec a' \wedge q \xrightarrow{a'} q''\big)}{q \xrightarrow{a}_\pi q'}$$

[6] The BIP engine implementing this semantics chooses one interaction at random, when faced with several enabled interactions.

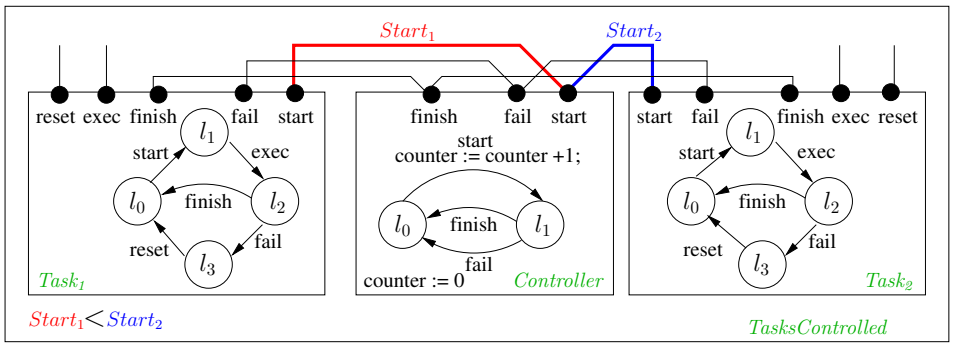


Fig. 3. A composite component in BIP

An interaction a is enabled in $\pi(C)$ whenever a is enabled in C and a is maximal according to π among the active interactions in C.

We adapt the notion of *maximal progress* to BIP systems. In BIP, the maximal progress property is expressed at the level of connectors. For a given connector γ, if one interaction $a \in \mathcal{I}(\gamma)$ is contained in another interaction $a' \in \mathcal{I}(\gamma)$, then the latter has a higher priority, unless there exists an explicit priority stating the contrary. Maximal progress is enforced by the BIP engine.

Definition 9 (Maximal Progress). *Given a connector γ and a priority model π, $\forall a, a' \in \mathcal{I}(\gamma)$: $(a \subset a') \wedge (a' \prec a \notin \pi) \Rightarrow a \prec a'$.*

Finally, we consider systems defined as a parallel composition of components together with an initial state.

Definition 10 (System). *A BIP system $\mathcal{S}$ is a pair $(B, Init)$ where B is a component and $Init \in B_1.L \times \ldots \times B_n.L$ is the initial state of B.*

For the sake of simpler notation, $Init$ designates both the initial state of the system at the syntax level and the initial state of the underlying LTS.

Example 2 (Composite component). Figure **??** shows an example of a composite component that consists of three atomic components ($Task_1$, $Task_2$, and $Controller$). The $Controller$ component is composed of a set of ports $\{start, finish, fail\}$, a set of locations $L = \{l_0, l_1\}$, a variable counter initialized to 0, and a set of transitions containing:

- $(l_0, start, \texttt{true}, [\text{counter:=counter+1}], l_1)$: from location l_0, the *start* port is enabled, the guard is `true`, when the transition is executed the component goes to location l_1 and the variable counter is incremented,
- $(l_1, finish, \texttt{true}, [\,], l_0)$: from location l_1, the *finish* port is enabled, the guard is `true`, when the transition is executed the component goes to location l_0 and no assignment is executed,
- $(l_1, fail, \texttt{true}, [\,], l_0)$: from location l_1, the *fail* port is enabled, the guard is `true`, when the transition is executed the component goes to location l_0 and no assignment is executed.

The two tasks have an identical model. A task is composed of a set of locations $\{l_0, l_1, l_2, l_3\}$, a set of ports $\{start, exec, finish, fail, reset\}$ and a set of transitions containing:

- $(l_0, start, \texttt{true}, [\,], l_1)$: from location l_0, the *start* port is enabled, the guard is `true`, when the transition is executed the component goes to location l_1 and no assignment is executed,
- $(l_1, exec, \texttt{true}, [\,], l_2)$: from location l_1, the *exec* port is enabled, the guard is `true`, when the transition is executed the component goes to location l_2 and no assignment is executed,
- $(l_2, \mathit{finish}, \texttt{true}, [\,], l_0)$: from location l_2, the *finish* port is enabled, the guard is `true`, when the transition is executed the component goes back to location l_0 and no assignment is executed,
- $(l_2, \mathit{fail}, \texttt{true}, [\,], l_3)$: from location l_2, the *fail* port is enabled, the guard is `true`, when the transition is executed the component goes to location l_3 and no assignment is executed,
- $(l_3, reset, \texttt{true}, [\,], l_0)$: from location l_3, the *reset* port is enabled, the guard is `true`, when the transition is executed the component goes back to location l_0 and no assignment is executed.

$Task_1$ and $Task_2$ must synchronize with $Controller$ to start/finish execution or to notify execution failure. Each $Task_i$ synchronizes with the controller using three connectors:

- $(\{Task_i.start, Controller.start\}, \{(Task_i.start, \texttt{false}), Controller.start, \texttt{false})\}, \texttt{true}, [\,])$,
- $(\{Task_i.\mathit{finish}, Controller.\mathit{finish}\}, \{(Task_i.\mathit{finish}, \texttt{false}), (Controller.\mathit{finish}, \texttt{false})\}, \texttt{true}, [\,])$,
- $(\{Task_i.\mathit{fail}, Controller.\mathit{fail}\}, \{(Task_i.\mathit{fail}, \texttt{false}), (Controller.\mathit{fail}, \texttt{false})\}, \texttt{true}, [\,])$.

For priorities, we consider that a task always has priority to start. This can be modeled easily in BIP by adding priorities between connectors: the connectors containing the port $start$ always have more priority than other connectors, that is, all interactions involved in $Start_1$ and $Start_2$ have more priority than the interactions involved in other connectors, for example, $\{Task_1.start, Controller.start\} \prec \{Task_2.start, Controller.start\}$.

The composite component $TasksControlled$ in this example is composed of a set of components $\{Task_1, Task_2, Controller\}$ and the connectors described above. The initial state is $Init = (Task_1.l_0, Controller.l_0, Task_2.l_0)$, following Definition **??**, the system is $(TasksControlled, Init)$.

4 An RV Framework for Component-Based Systems

We adapt classical RV frameworks dedicated to monitoring of sequential monolithic programs to CBS in general, and, to BIP systems in particular. We consider $B = \pi(\Gamma(\{B_1, \ldots, B_n\}))$, i.e., a priority model π over a composite component $\Gamma(\{B_1, \ldots, B_n\})$, whose runtime semantics is $\pi(C)$, defined by an LTS $(Q, A, \longrightarrow_\pi)$ as introduced in Definitions **??** and **??**.

4.1 Specifications for Component-Based Systems

For CBS, we consider state-based specifications to express some desired behavior. We do not assume any particular specification formalism. We require the formalism to express a subset of the possible linear-time behaviors of CBS. In order to make our approach as general as possible, we only describe the events of the possible specification language. We also assume the existence of a monitor synthesis algorithm from this specification formalism (see Section **??**). For this purpose, the existing solutions (e.g., [**?**,**?**,**?**,**?**,**?**]) provided by the research efforts in RV can be easily adapted.

We follow a classical approach where events are built over a set of atomic propositions AP. Intuitively, an atomic proposition is a Boolean expression over the states of the components (e.g., "in the component B_1, the variable x should be positive if in the component B_2 the variable y is negative"). More formally, an event of $\pi(C)$ is defined as a state formula over the atomic propositions expressed on components involved in $\pi(C)$. Let AP denote the set of atomic propositions defined with the following grammar:

$$\begin{aligned}
\text{Atom} ::= {} & \text{component}_1.\text{var}_1 == \text{component}_2.\text{var}_2 \mid \text{component.var} == \text{val} \\
& \mid \text{component.var} \geq \text{val} \mid \text{component.loc} == \text{a_location} \mid \text{component.port} == \text{a_port} \\
\text{component.var} ::= {} & x \in \cup_{i\in[1,n]} B_i.X \\
\text{val} ::= {} & v \in \text{Data} \\
\text{a_location} ::= {} & s \in \cup_{i\in[1,n]} B_i.L \\
\text{a_port} ::= {} & p \in \cup_{i\in[1,n]} B_i.P
\end{aligned}$$

An atomic proposition consists in a comparison of the values of some variables, the current location, or the port that is on the last executed transition.

Let Σ denote the set of events defined with the following grammar:

$$\text{Event} ::= \text{Event} \vee \text{Event} \mid \text{Event} \wedge \text{Event} \mid \text{Event} \Rightarrow \text{Event} \mid \neg\ \text{Event} \mid \text{Atom}$$

In the remainder of this article, we suppose that all the atomic propositions appearing in the property affect its truth-value.[7] We use $Prop : \Sigma \to 2^{AP}$ for the set of atomic propositions used in an event $e \in \pi(C)$. More formally, $Prop$ is defined inductively by using the following rules:

$$\begin{aligned}
Prop(\text{component}_1.\text{var}_1 == \text{component}_2.\text{var}_2) &\stackrel{\text{def}}{=} \{\text{component}_1.\text{var}_1 == \text{component}_2.\text{var}_2\}, \\
Prop(\text{component.var} == \text{val}) &\stackrel{\text{def}}{=} \{\text{component.var} == \text{val}\}, \\
Prop(\text{component.var} \geq \text{val}) &\stackrel{\text{def}}{=} \{\text{component.var} \geq \text{val}\}, \\
Prop(\text{component.loc} == \text{a_location}) &\stackrel{\text{def}}{=} \{\text{component.loc} == \text{a_location}\}, \\
Prop(\text{component.port} == \text{a_port}) &\stackrel{\text{def}}{=} \{\text{component.port} == \text{a_port}\}, \\
Prop(e_1 \vee e_2) &\stackrel{\text{def}}{=} Prop(e_1) \cup Prop(e_2), \\
Prop(e_1 \wedge e_2) &\stackrel{\text{def}}{=} Prop(e_1) \cup Prop(e_2), \\
Prop(e_1 \Rightarrow e_2) &\stackrel{\text{def}}{=} Prop(e_1) \cup Prop(e_2), \\
Prop(\neg e) &\stackrel{\text{def}}{=} Prop(e).
\end{aligned}$$

For $ap \in Prop(e)$, $used(ap)$ is the list of pairs formed by the components and the variables (or locations or ports) that are used to define ap. The expression $used(ap)$ is defined using a *pattern-matching*:

$$\begin{aligned}
&\texttt{used(ap)} = \texttt{match(ap) with} \\
&\quad \mid \texttt{component}_1\texttt{.var}_1 == \texttt{component}_2\texttt{.var}_2 \to (\texttt{component}_1, \texttt{var}_1) \cdot (\texttt{component}_2, \texttt{var}_2) \\
&\quad \mid \texttt{component.var} == \texttt{val} \to (\texttt{component}, \texttt{var}) \\
&\quad \mid \texttt{component.var} \geq \texttt{val} \to (\texttt{component}, \texttt{var}) \\
&\quad \mid \texttt{component.loc} == \texttt{a_location} \to (\texttt{component}, \texttt{loc}) \\
&\quad \mid \texttt{component.port} == \texttt{a_port} \to (\texttt{component}, \texttt{port})
\end{aligned}$$

[7] Otherwise, some simplification of the specification shall be performed beforehand. For instance, such simplification should rule out events of the form $a \vee \neg a$ where $a \in$ Atom.

Example 3 (Atomic propositions, events). Suppose we want to monitor the execution ordering of the tasks involved in the composite component introduced in Example **??**. Note that such kind of properties is very difficult to enforce with priorities. To verify such properties, we can observe the execution of the transitions involving the start ports of the two components. The set of atomic propositions of such a property is $\{Task_1.\mathrm{port} == \mathrm{start}, Task_2.\mathrm{port} == \mathrm{start}\}$. The set of events is equal to the set of atomic propositions. Moreover, we have $used(Task_1.\mathrm{port} == \mathrm{start}) = (Task_1, \mathrm{port})$ and $used(Task_2.\mathrm{port} == \mathrm{start}) = (Task_2, \mathrm{port})$.

4.2 Verification Monitors

A monitor is a procedure that consumes events fed by a BIP system and producing an appraisal on the sequence of events read so far. We follow a general approach in which verification monitors are deterministic finite-state machines that produce a sequence of truth-values (a sequence of verdicts) in an expressive 4-valued truth-domain $\mathbb{B}_4 \stackrel{\text{def}}{=} \{\bot, \bot_c, \top_c, \top\}$, as introduced in [**?**] and used in [**?**]. $\mathbb{B}_4$ consists of the possible evaluations of a sequence of events and its possible futures relatively to the specification used to generate the monitor:

- The truth-value $\top_c$ (resp. $\bot_c$) denotes "currently true" (resp. "currently false") and expresses the satisfaction (resp. violation) of the specification "if the system execution stops here".
- The truth-value $\top$ (resp. $\bot$) is a definitive verdict denoting the satisfaction (resp. violation) of the specification: the monitor can be stopped.

Remark 1 (Other verdict domains and monitorable properties). As demonstrated in [**?**], several more restricted verdict domains can be derived from $\mathbb{B}_4$ so as to fit a given specification language. The set of verdicts (used for monitoring) determines the so called *monitorable properties* (cf. [**?**]). Using the four-valued domain $\mathbb{B}_4$ allows to monitor any linear-time specification over finite executions, as demonstrated in [**?**]. Using a less expressive verdict domain such as $\mathbb{B}_3$ (defined in [**?**]) is also possible up to a restriction on the monitorable properties (cf. [**?**,**?**]). In this paper, we present the monitoring framework with the most general verdict domain $\mathbb{B}_4$ and thus do not consider any restriction on the set of monitorable properties. See [**?**,**?**] for more details.

We define the notion of monitor for a specification defined relatively to a set of events Σ expressed on a composite component. Monitors are deterministic Moore (finite-state) machines emitting a verdict on each state.

Definition 11 (Monitor). *A* monitor $\mathcal{A}$ *is a tuple* $(\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\mathrm{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$. *The finite set* $\Theta^{\mathcal{A}}$ *denotes the control states and* $\theta^{\mathcal{A}}_{\mathrm{init}} \in \Theta^{\mathcal{A}}$ *is the initial state. The complete function* $\longrightarrow_{\mathcal{A}}$: $\Theta^{\mathcal{A}} \times \Sigma \to \Theta^{\mathcal{A}}$ *is the transition function. In the following we abbreviate* $\longrightarrow_{\mathcal{A}} (\theta, a) = \theta'$ *by* $\theta \stackrel{a}{\longrightarrow}_{\mathcal{A}} \theta'$. *The function* $ver^{\mathcal{A}} : \Theta^{\mathcal{A}} \to \mathbb{B}_4$ *is an output function, producing verdicts (i.e., truth-values) in* $\mathbb{B}_4$ *from control states.*

Such monitors are independent from any specification formalism used to generate them and are able to check any specification expressing a linear temporal specification [**?**]. Intuitively, runtime verification of

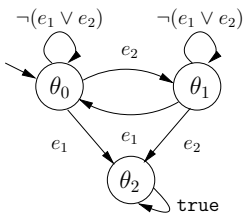


Fig. 4. The monitor $\mathcal{A}_{alt}$ for the alternation of task executions

a specification with such monitors works as follows. An execution sequence is processed in a lock-step manner. On each received event, the monitor produces an appraisal on the sequence read so far. For a formal presentation of the semantics of the monitor and a formal definition of sequence checking, we refer to [**?**].

Example 4 (Monitor). Let us write a monitor $\mathcal{A}_{alt}$ to runtime verify the behavior of the composite component introduced in Example **??**. The considered property states that the execution of $Task_1$ and $Task_2$ should alternate strictly, starting with $Task_2$. The monitor $\mathcal{A}_{alt}$ is defined by the 5-tuple $(\Theta^{\mathcal{A}_{alt}}, \theta^{\mathcal{A}_{alt}}_{\text{init}}, \Sigma^{\mathcal{A}_{alt}}, \longrightarrow_{\mathcal{A}_{alt}}, \mathbb{B}_4, ver^{\mathcal{A}_{alt}})$. A graphical representation of $\mathcal{A}_{alt}$ is depicted in Fig. **??**. The set of states is $\Theta^{\mathcal{A}_{alt}} = \{\theta_0, \theta_1, \theta_2\}$. The initial state is $\theta^{\mathcal{A}_{alt}}_{\text{init}} = \theta_0$. The set of events is $\Sigma^{\mathcal{A}_{alt}} = \{Task_1.\text{port} == \text{start}, Task_2.\text{port} == \text{start}\}$ with $e_1 \stackrel{\text{def}}{=} Task_1.\text{port} == \text{start}$ and $e_2 \stackrel{\text{def}}{=} Task_2.\text{port} == \text{start}$. The set of transitions is $\longrightarrow_{\mathcal{A}_{alt}} = \{(\theta_0, e_1, \theta_2), (\theta_0, e_2, \theta_1), (\theta_0, \neg(e_1 \vee e_2), \theta_0), (\theta_1, e_1, \theta_0), (\theta_1, e_2, \theta_2), (\theta_1, \neg(e_1 \vee e_2), \theta_1), (\theta_2, \texttt{true}, \theta_2)\}$. Only two verdicts from $\mathbb{B}_4$ are needed to monitor this property: $\top_c$ and $\bot$. The verdicts associated to states through the verdict function are as follows: $ver^{\mathcal{A}_{alt}}(\theta_0) = ver^{\mathcal{A}_{alt}}(\theta_1) = \top_c$ and $ver^{\mathcal{A}_{alt}}(\theta_2) = \bot$, i.e., the property is "currently true" when the monitor is in states θ_0 and θ_1 and (definitely) false when $\mathcal{A}_{alt}$ is in θ_2.

In the remainder, we consider a monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$.

4.3 Runs and Traces of BIP Systems

Runtime monitors observe the evolving state of the system by processing a so-called *run* of the system. Intuitively, a run is the sequence of all states reached by a BIP system during its execution. However, when monitoring a given property of a system, some information related to the current state of a system can be safely discarded to evaluate the property. In this subsection, we present the notion of run of a CBS, and how, using the vocabulary of the monitored property, we can discard some information in the run. Such an abstraction of a run is called a *trace*.

Runs of BIP systems. Each state $q \in Q$ in the LTS of a component can be seen as an environment that maps variables used in the specification over an alphabet Σ to values. In each atomic component, we introduce two variables loc and $port$ to represent the current location and the last executed port, respectively. Then, to construct the universe of possible environments, we build the set of functions that are consistent merges (using operator $\biguplus$ introduced in Section **??**) of the functions from variables to data, from the variable $B_i.loc$ to $B_i.L$, from the variable $port$ to $B_i.P \cup \texttt{null}$, for each atomic component $B_i, i \in [1, n]$.

Definition 12 (Environments in a component). *The universe of possible environments for $\pi(C)$ is*

$$Env \stackrel{\text{def}}{=} \biguplus \Big\{ \begin{array}{l} \big[\cup_{i\in[1,n]} B_i.X \to Data \big], \\ \quad \big[B_1.loc \to (B_1.L)], \ldots, [B_n.loc \to (B_n.L)\big], \\ \quad \big[B_1.port \to B_1.P \cup \{\texttt{null}\}\big], \ldots, \big[B_n.port \to B_n.P \cup \{\texttt{null}\}\big] \end{array} \Big\}$$

where $B_i.loc$ and $B_i.port$ are variables containing a location name and a port name for each atomic component B_i, respectively. The environment defined by a state $q = (q_1, \ldots, q_n)$, where $q_i = (l_i, v_i, p_i)$ for each $i \in [1, n]$, is $[\![q]\!] \in Env$ such that:

$$[\![q]\!] \stackrel{\text{def}}{=} \cup_{i\in[1,n]} \Big(\big(\cup_{i\in[1,n]} \{B_i.loc \mapsto l_i\}\big) \cup \big(\cup_{x_i \in B_i.X} \{x_i \mapsto v_i\}\big) \cup \big(\cup_{i\in[1,n]} \{B_i.port \mapsto p_i\}\big)\Big).$$

After an interaction bringing the component in a state q, an event e is fired if the state-formula associated to e holds, noted $q \vDash e$, i.e., when e evaluates to true in $[\![q]\!]$. Formally, q satisfies e (denoted by $q \vDash e$), as defined by the following rules:

$$\begin{aligned} q \vDash (\text{component}_1.\text{var}_1 == \text{component}_2.\text{var}_2) &\Leftrightarrow [\![q]\!](\text{component}_1.\text{var}_1) == [\![q]\!](\text{component}_2.\text{var}_2), \\ q \vDash (\text{component.var} == \text{val}) &\Leftrightarrow [\![q]\!](\text{component.var}) == \text{val}, \\ q \vDash (\text{component.var} \geq \text{val}) &\Leftrightarrow [\![q]\!](\text{component.var}) \geq \text{val}, \\ q \vDash (\text{component.loc} == \text{a\_location}) &\Leftrightarrow [\![q]\!](\text{component.loc}) == \text{a\_location}, \\ q \vDash (\text{component.port} == \text{a\_port}) &\Leftrightarrow [\![q]\!](\text{component.port}) == \text{a\_port}, \\ q \vDash (e_1 \vee e_2) &\Leftrightarrow (q \vDash e_1) \vee (q \vDash e_2), \\ q \vDash (e_1 \wedge e_2) &\Leftrightarrow (q \vDash e_1) \wedge (q \vDash e_2), \\ q \vDash (e_1 \Rightarrow e_2) &\Leftrightarrow (q \vDash e_1) \Rightarrow (q \vDash e_2), \\ q \vDash (\neg e) &\Leftrightarrow \neg(q \vDash e). \end{aligned}$$

Note that, after reaching a state of the LTS corresponding to the runtime behavior of a BIP component, it is always possible to determine whether an event is fired or not by checking whether the corresponding state-formula holds or not.

Some constraints on the monitors. For the monitor to properly evaluate events in states, we impose two constraints on their events and transition function, called *readiness* and *determinism*. These constraints intuitively state that the events labelling the transitions of an abstract monitor are such that exactly one transition will be fired. This condition is expressed using the Boolean conditions corresponding to the events of the automaton. The following definition formalizes these properties.

Definition 13 (Readiness and determinism of monitors). *The readiness and determinism properties of a monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$ are defined as follows. For $\theta \in \Theta^{\mathcal{A}}$, let $events(\mathcal{A}, \theta) = \{e \in \Sigma \mid \theta \xrightarrow{e}\}$. Then*

$$\begin{aligned} \forall \theta \in \Theta^{\mathcal{A}}, \forall q \in Env: \ & q \models \bigvee\nolimits_{e \in events(\mathcal{A},\theta)} e && \textit{(readiness)} \\ & \wedge \forall e_1, e_2 \in events(\mathcal{A}, \theta) : e_1 \neq e_2 \Rightarrow q \not\models e_1 \wedge e_2 && \textit{(determinism)} \end{aligned}$$

Readiness means that, in any state, the disjunction of Boolean conditions corresponding to the possible events evaluates to `true`. Determinism means that, in any state, each pairwise conjunction of Boolean conditions corresponding to the possible events evaluates to `false`. Readiness and determinism ensure that in a given state, exactly one transition is fired when the monitor receives any event. Thus, given a behavior of the underlying system, only one possible verdict is dictated by the monitor.[8]

Monitoring a run of a composite component. We present the notion of *run* of a composite component and how the run is monitored.

Definition 14 (Run of a composite component). *A run of length m of a system $(B, Init)$ whose runtime semantics is $\pi(C) = (Q, A, \longrightarrow_\pi)$ is the sequence of environments $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ such that: q^0 = $Init$, and, $\forall i \in [0, m-1] : q^i \in Q \wedge \exists a_i \in A : q^i \xrightarrow{a_i}_\pi q^{i+1}$.*

Example 5 (Run of $TasksControlled$). Consider the following execution scenario for the composite component $TasksControlled$: $Task_2$ get executed, then $Task_1$ get executed and fails, then $Task_2$ get executed twice. The desired property (stating that the execution of $Task_1$ and $Task_2$ should alternate strictly, starting with $Task_2$) is thus violated by this execution. This execution yields the run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^{11}]\!]$, where:

$$
\begin{aligned}
[\![q^0]\!] &= \{Task_1.loc = l_0, Task_1.port = \texttt{null}\} \cup \{Controller.loc = l_0, Controller.port = \texttt{null}\} \\
&\quad \cup \{Task_2.loc = l_0, Task_2.port = \texttt{null}\} \\
[\![q^1]\!] &= \{Task_1.loc = l_0, Task_1.port = \texttt{null}\} \cup \{Controller.loc = l_1, Controller.port = start\} \\
&\quad \cup \{Task_2.loc = l_1, Task_2.port = start\} \\
[\![q^2]\!] &= \{Task_1.loc = l_0, Task_1.port = \texttt{null}\} \cup \{Controller.loc = l_1, Controller.port = start\} \\
&\quad \cup \{Task_2.loc = l_2, Task_2.port = exec\} \\
[\![q^3]\!] &= \{Task_1.loc = l_0, Task_1.port = \texttt{null}\} \cup \{Controller.loc = l_0, Controller.port = finish\} \\
&\quad \cup \{Task_2.loc = l_0, Task_2.port = finish\} \\
[\![q^4]\!] &= \{Task_1.loc = l_1, Task_1.port = start\} \cup \{Controller.loc = l_1, Controller.port = start\} \\
&\quad \cup \{Task_2.loc = l_0, Task_2.port = finish\} \\
[\![q^5]\!] &= \{Task_1.loc = l_2, Task_1.port = exec\} \cup \{Controller.loc = l_1, Controller.port = start\} \\
&\quad \cup \{Task_2.loc = l_0, Task_2.port = finish\} \\
[\![q^6]\!] &= \{Task_1.loc = l_3, Task_1.port = fail\} \cup \{Controller.loc = l_0, Controller.port = fail\} \\
&\quad \cup \{Task_2.loc = l_0, Task_2.port = finish\} \\
[\![q^7]\!] &= \{Task_1.loc = l_3, Task_1.port = fail\} \cup \{Controller.loc = l_1, Controller.port = start\} \\
&\quad \cup \{Task_2.loc = l_1, Task_2.port = start\} \\
[\![q^8]\!] &= \{Task_1.loc = l_0, Task_1.port = reset\} \cup \{Controller.loc = l_1, Controller.port = start\} \\
&\quad \cup \{Task_2.loc = l_1, Task_2.port = start\} \\
[\![q^9]\!] &= \{Task_1.loc = l_0, Task_1.port = reset\} \cup \{Controller.loc = l_1, Controller.port = start\} \\
&\quad \cup \{Task_2.loc = l_2, Task_2.port = exec\}
\end{aligned}
$$

[8] This is a reasonable and usual hypothesis in runtime verification since one expects to characterize the behavior of an implementation in a deterministic way. Moreover, these two constraints are easily and naturally ensured by a monitor generation tool using specification written in a higher-level formalism as input. Finally, note that readiness corresponds to the standard concept of completeness in automata theory.

$$[\![q^{10}]\!] = \{Task_1.loc = l_0, Task_1.port = reset\} \cup \{Controller.loc = l_0, Controller.port = finish\} \cup \{Task_2.loc = l_0, Task_2.port = finish\}$$
$$[\![q^{11}]\!] = \{Task_1.loc = l_0, Task_1.port = reset\} \cup \{Controller.loc = l_1, Controller.port = start\} \cup \{Task_2.loc = l_1, Task_2.port = start\}$$

Definition 15 (Monitoring a run of a system). *The verdict* $[\![\mathcal{A}]\!](q^0 \cdot q^1 \cdots q^m)$ *stated by* $\mathcal{A}$ *for a run* $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ *is* $ver^{\mathcal{A}}(\theta^m)$ *where* $\forall i \in [0, m-1] : \theta_i \xrightarrow{e}_{\mathcal{A}} \theta_{i+1}$ *and* e *is the unique event*[9] *in* θ_i *such that* $q^{i+1} \models e$, $\theta_i \in \Theta^{\mathcal{A}}$ *and* $\theta_0 = \theta^{\mathcal{A}}_{\text{init}}$.

Example 6 (Monitoring a run of TasksControlled). Monitoring the run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^{11}]\!]$ of Example **??** with the monitor $\mathcal{A}_{alt}$ introduced in Example **??**, yields a verdict for each of the prefixes of $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^{11}]\!]$. More precisely, we have $\forall i \in [0, 10] : [\![\mathcal{A}_{alt}]\!](q^0 \cdots q^i) = \top_c$ and $[\![\mathcal{A}_{alt}]\!](q^0 \cdots q^{11}) = \bot$.

Building a trace from a run. One of the current challenges in RV is to alleviate the performance impact on the target program. We tackle this challenge by minimizing the information sent to the monitor. Making the monitor processing the run of the target program directly would yield a prohibitive overhead. Our proposal is to send to the monitor only a relevant abstraction of the run, called a *trace*. Intuitively, given a run, the obtained trace is an abstraction that permits to evaluate the specification as if the run was not abstracted, filtering many irrelevant events that are guaranteed to be irrelevant for the monitor. Given $Spec(\Sigma)$, a specification defined over a vocabulary of events Σ, we design an abstraction function $\downarrow^{\Sigma}_{\alpha}$ building this minimal abstraction. We thus define a notion of informativeness of environments built from states. Intuitively, an environment ρ_1 is less informative than an environment ρ_2 if it has less variables defined, i.e., $\rho_1 \sqsubseteq \rho_2$ if $Dom(\rho_1) \subseteq Dom(\rho_2)$ and $\forall x \in Dom(\rho_1) : \rho_1(x) = \rho_2(x)$. When monitoring a BIP system our aim will be to instrument it so that the least informative environment is automatically built. Moreover, according to property evaluation, monitoring the instrumented system should be equivalent to monitoring the system with the global state.

Definition 16 (Abstraction function). *The abstraction function* $\downarrow^{\Sigma}_{\alpha}$: $Q \to Env$ *is the function such that:* $\forall q \in Q, \forall x \in Dom([\![q]\!])$:

$$\downarrow^{\Sigma}_{\alpha}(q)(B_i.x) = \begin{cases} [\![q]\!](B_i.x) & \text{if } \exists e \in \Sigma, \exists ap \in Prop(e) : (B_i, x) \in used(ap), \\ undef & \text{otherwise.} \end{cases}$$

It turns out that it might become impossible to evaluate some atomic propositions in some abstracted environment: when an abstraction function "erases" a piece of information needed to evaluate an event e, it becomes impossible to determine whether a state q satisfies e. Intuitively, the law of excluded middle does not hold with some environments. More formally:

$$\forall q \in Q, \forall e \in \Sigma : \big(\exists ap \in Prop(e) : (B_i, x) \in used(ap) \wedge [\![q]\!](B_i.x) = undef\big) \Rightarrow \neg(q \vDash e) \wedge \neg(q \nvDash e).$$

[9] This event is unique because of determinism (see Definition **??**).

However, this situation does not arise with the abstraction function defined in Definition **??**. Indeed, the proposed abstraction function preserves event evaluation because it is *sound* and *complete*. Soundness states that the abstracted evaluations are the same as the concrete evaluations. Completeness states that evaluation of all specification events remains possible: abstraction does not erase the needed information from the environment. More formally, soundness and completeness are expressed by the following property.

Property 1 (Abstraction preserves event evaluation). The previous abstraction function adheres to the two following principles:

- soundness: $\forall e \in \Sigma, \forall q \in Q : \big(\downarrow_\alpha^\Sigma (q) \vDash e \Rightarrow q \vDash e \big) \wedge \big(\downarrow_\alpha^\Sigma (q) \nvDash e \Rightarrow q \nvDash e \big)$,
- completeness: $\forall e \in \Sigma, \forall q \in Q : \big(q \vDash e \Rightarrow \downarrow_\alpha^\Sigma (q) \vDash e \big) \wedge \big(q \nvDash e \Rightarrow \downarrow_\alpha^\Sigma (q) \nvDash e \big)$.

Proof. Soundness is a straightforward consequence of the definition of the abstraction function. The proof of the completeness property is done by induction on the structure of the event.

- Induction basis. Consider an event $e \in AP$.
  - Let us treat the case where e is of the form component$_1$.var$_1$ == component$_2$.var$_2$. That is, $used(e)$ = (component$_1$,var$_1$) · (component$_2$,var$_2$). Moreover, because $e \in \Sigma$ then for all q, we have: $\downarrow_\alpha^\Sigma (q)$(component$_1$.var$_1$) = $[\![q]\!]$(component$_1$.var$_1$) and $\downarrow_\alpha^\Sigma (q)$(component$_2$.var$_2$) = $[\![q]\!]$(component$_2$.var$_2$) (see Definition **??**). If $q \vDash e$, it means that $[\![q]\!]$(component$_1$.var$_1$) == $[\![q]\!]$(component$_2$.var$_2$), and then $\downarrow_\alpha^\Sigma (q)$(component$_1$.var$_1$) == $\downarrow_\alpha^\Sigma (q)$(component$_2$.var$_2$) which implies that $\downarrow_\alpha^\Sigma (q) \vDash e$. If $q \nvDash e$, it means that $[\![q]\!]$(component$_1$.var$_1$) $\neq$ $[\![q]\!]$(component$_2$.var$_2$). Consequently, $\downarrow_\alpha^\Sigma (q)$(component$_1$.var$_1$) $\neq\downarrow_\alpha^\Sigma (q)$(component$_2$.var$_2$) and then $\downarrow_\alpha^\Sigma (q) \nvDash e$.
  - Let us treat the case where e is of the form component.var == val. That is, $used(e)$ = (component,var). Moreover, because $e \in \Sigma$ then for all q, we have: $\downarrow_\alpha^\Sigma (q)$(component.var) = $[\![q]\!]$(component.var) (see Definition **??**). If $q \vDash e$, it means that $[\![q]\!]$(component.var) == val, and then $\downarrow_\alpha^\Sigma (q)$(component.var) == val which implies that $\downarrow_\alpha^\Sigma (q) \vDash e$. If $q \nvDash e$, it means that $[\![q]\!]$(component.var) $\neq$ val. Consequently, $\downarrow_\alpha^\Sigma (q)$(component.var) $\neq$ val and then $\downarrow_\alpha^\Sigma (q) \nvDash e$.
  - The same principle can be followed in the cases where e is of the form: component.var $\geq$ val, or component.loc == a_location, and component.port == a_port.
- Induction step. Let us consider two events $e_1, e_2 \in \mathrm{Event}$ such that the completeness property holds. We consider now an event $e \in \mathrm{Event}$ built on e_1 and e_2. We distinguish several cases according to how e is built.
  - Let us treat the case where e is of the form $e_1 \vee e_2$. If $(q \vDash e) \Rightarrow (q \vDash e_1) \vee (q \vDash e_2)$. As $Prop(e_1) \subseteq Prop(e)$ and $Prop(e_2) \subseteq Prop(e)$, the induction hypothesis gives $(\downarrow_\alpha^\Sigma (q) \vDash e_1) \vee (\downarrow_\alpha^\Sigma (q) \vDash e_2) \Rightarrow \downarrow_\alpha^\Sigma (q) \vDash (e_1 \wedge e_2)$. The same principle is applied for the case where $q \nvDash e$
  - The same principle can be followed in the cases where e is of the form: $e_1 \wedge e_2$, $e_1 \Leftrightarrow e_2$, or $\neg e_1$.

Definition 17 (Trace of a composite component). *The trace defined from a run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ through an abstraction function $\downarrow_\alpha^\Sigma$ is the sequence of environments defined as $\downarrow_\alpha^\Sigma (q^0) \cdot \downarrow_\alpha^\Sigma (q^1) \cdots \downarrow_\alpha^\Sigma (q^m)$.*

Example 7 (Trace of a composite component). From the run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^{11}]\!]$ described in Example **??**, through the abstraction function $\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}$, we obtain the trace $\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^0) \cdot \downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^1) \cdots \downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^{11})$ where:

$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^0) = \{Task_1.port = \texttt{null}\} \cup \{Task_2.port = \texttt{null}\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^1) = \{Task_1.port = \texttt{null}\} \cup \{Task_2.port = start\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^2) = \{Task_1.port = \texttt{null}\} \cup \{Task_2.port = exec\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^3) = \{Task_1.port = \texttt{null}\} \cup \{Task_2.port = finish\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^4) = \{Task_1.port = start\} \cup \{Task_2.port = finish\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^5) = \{Task_1.port = exec\} \cup \{Task_2.port = finish\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^6) = \{Task_1.port = fail\} \cup \{Task_2.port = finish\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^7) = \{Task_1.port = fail\} \cup \{Task_2.port = start\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^8) = \{Task_1.port = reset\} \cup \{Task_2.port = start\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^9) = \{Task_1.port = reset\} \cup \{Task_2.port = exec\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^{10}) = \{Task_1.port = reset\} \cup \{Task_2.port = finish\}$,
$\downarrow_\alpha^{\Sigma_{\mathcal{A}_{act}}}(q^{11}) = \{Task_1.port = reset\} \cup \{Task_2.port = start\}$.

The notion of trace evaluation by a monitor directly follows from the notion of run evaluation. Moreover, the following theorem, which is a direct consequence of Property **??**, states that, for runtime verification, there is no difference regarding property evaluation to process the trace instead of the run.

Theorem 1 (Trace evaluation vs run evaluation by a monitor). *For $\mathcal{A}$ defined on Σ, the abstraction function $\downarrow_\alpha^\Sigma$, and a run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$, we have:*

$$[\![\mathcal{A}]\!]\big([\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]\big) = [\![\mathcal{A}]\!]\big(\downarrow_\alpha^\Sigma(q^0) \cdot \downarrow_\alpha^\Sigma(q^1) \cdots \downarrow_\alpha^\Sigma(q^m)\big).$$

Proof. By induction on the length of the trace and using Property **??**.

In the next section, we will instrument BIP systems in such a way that, given a specification, the minimal abstraction function (information-wise) is dynamically generated.

5 Verifying the Runtime Behavior of BIP Systems

This section presents how we instrument and integrate an abstract monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$ for some property into a BIP system $\big(\Gamma(\{B_1, \ldots, B_n\}), (l_0^1, \ldots, l_0^n)\big)$ made of a composite component $\Gamma(\{B_1, \ldots, B_n\})$, a priority model π where the initial locations of the atomic components $B_1, \ldots, B_n$ are $l_0^1, \ldots, l_0^n$, respectively.

We propose several transformations of the initial BIP system. Our transformations result in a BIP system where the property is automatically runtime checked against the execution of the system. More precisely, the work-flow proceeds as follows (see Fig. **??**):

1. From the input abstract monitor we extract the list of components and their corresponding variables used by the monitor (Section **??**).

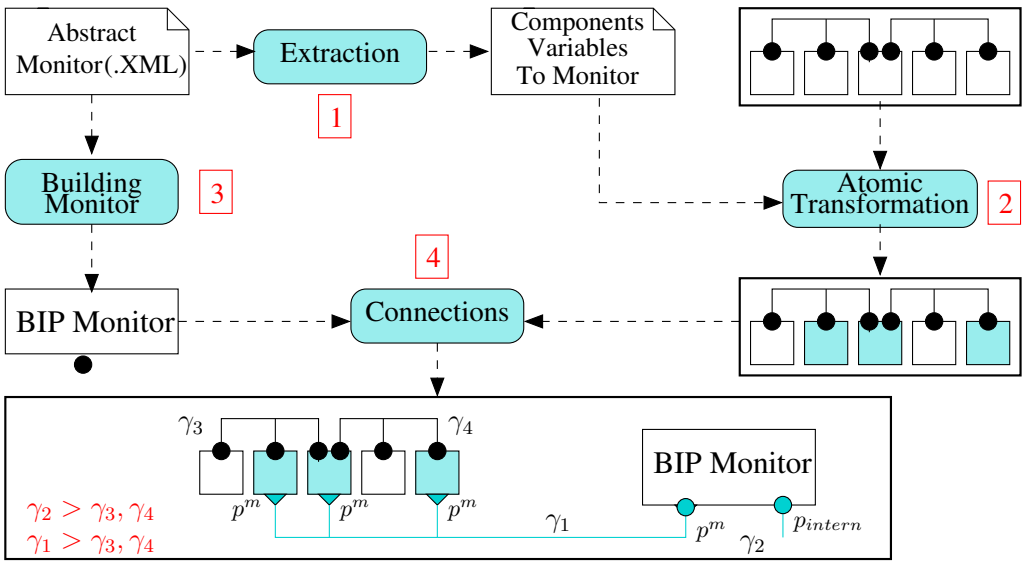


Fig. 5. Overview of the work-flow

2. For each component B_i where $i \in [1, n]$, and its corresponding variables extracted from the monitor we instrument the selected components so as to observe these variables (Section **??**).
3. From the monitor we generate the corresponding atomic component: the initial monitor is translated in BIP. This component can receive the state of the underlying system and then process the information to reach a verdict. Then, we add the generated component to the input composite component (Section **??**).
4. Finally, we add the new connections between the instrumented atomic components and the monitor in BIP (Section **??**).

5.1 Extraction of the Information Needed

The first step is to retrieve from the monitor the set of components and their corresponding variables that should be monitored. For each selected component, transitions are instrumented to observe the just needed set of variables. For a specification expressed over $\Sigma(\pi(\Gamma(\{B_1, \ldots, B_n\})))$ and its monitor, $comp(\Sigma)$ is the subset of $\{B_i \mid i \in [1, n]\}$ corresponding to the set of components that should be monitored. We also define $occur(\Sigma)$ to be the subset of $\{B_i.loc \mid i \in [1, n]\} \cup \{B_i.port \mid i \in [1, n]\} \cup \cup_{i \in [1,n]} B_i.X$ denoting the set of variables used in the specification. Then from $occur(\Sigma)$, we sort the variables according to the component B_i (where $B_i \in comp(\Sigma)$) that are related to: $c\_v \in \big[[1, n] \rightarrow occur(\Sigma)\big]$ such that $c\_v(i)$ is the set of variables related to component B_i. Formally:

$$comp(\Sigma) = \{B_i \mid \exists e \in \Sigma, \exists ap \in Prop(e), \exists x \in B_i.X : (B_i, x) \in used(ap)\} \quad (1)$$

$$occur(\Sigma) = \{B_i.x \mid \exists e \in \Sigma, \exists ap \in Prop(e) : (B_i, x) \in used(ap) \wedge x \in B_i.X\} \quad (2)$$

$$\forall i \in [1, n] : c\_v(i) = \{B.x \in occur(\Sigma) \mid B = B_i\} \quad (3)$$

5.2 Instrumentation of Atomic Components

For a composite component $\Gamma(\{B_1, \ldots, B_n\})$, we transform each atomic component B_i, $i \in [1, n]$, so that B_i is able to interact with the monitor, if necessary.

Definition 18 (Instrumenting atomic components). *Given $B = (P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ such that $B = B_i \in \{B_1, \ldots, B_n\}$ for some $i \in [1, n]$, we define a new atomic component*

$$B^m = \begin{cases} B & \text{if } B \notin comp(\Sigma) \\ (P^m, L^m, T^m, X^m, \{g_\tau\}_{\tau \in T^m}, \{f_\tau\}_{\tau \in T^m}) & \text{otherwise} \end{cases}$$

where, $(P^m, L^m, T^m, X^m, \{g_\tau\}_{\tau \in T^m}, \{f_\tau\}_{\tau \in T^m})$ is defined as follows:

- $X^m = X \cup \{loc \mid B_i.loc \in c\_v(i)\} \cup \{port \mid B_i.port \in c\_v(i)\}$ *where loc and port are initialized to l_0^i and* `null`*, respectively;*
- $P^m = P \cup \{p^m[c\_v(i)]\}$,
- $L^m = L \cup \{l_\tau\}_{\tau \in inst(T)}$*, where, $inst(T)$ is defined as follows:*

$$inst(T) = \begin{cases} T & \text{if } \{B_i.loc, B_i.port\} \cap c\_v(i) \neq \emptyset \\ \{\tau \in T \mid c\_v(i) \cap var(f_\tau) \neq \emptyset\} & \text{otherwise} \end{cases}$$

  where, $var(f_\tau) = \{x \in X \mid x := f^x(X) \in f_\tau\}$;
- $T^m = T \setminus inst(T) \cup \bigcup_{\tau \in inst(T)} \{in(\tau), out(\tau)\}$*, where, for a given $\tau = (l, p, g_\tau, f_\tau, l')$, we simultaneously generate $in(\tau)$ and $out(\tau)$ as follows:*
  - $in(\tau) = (l, p, g_\tau, f_{in(\tau)}, l_\tau)$*, where,*

$$f_{in(\tau)} = \begin{cases} f_\tau & \text{if } B_i.loc \notin c\_v(i) \wedge B_i.port \notin c\_v(i) \\ f_\tau; [loc := \text{“}l'\text{”}] & \text{if } B_i.loc \in c\_v(i) \wedge B_i.port \notin c\_v(i) \\ f_\tau; [port := \text{“}p\text{”}] & \text{if } B_i.loc \notin c\_v(i) \wedge B_i.port \in c\_v(i) \\ f_\tau; [loc := \text{“}l'\text{”}; port := \text{“}p\text{”}] & \text{if } B_i.loc \in c\_v(i) \wedge B_i.port \in c\_v(i) \end{cases}$$

  - $out(\tau) = (l_\tau, p^m, \texttt{true}, f_{out(\tau)}, l')$*, where $f_{out(\tau)} = [\,]$.*

We denote $B^m = \mathrm{Instrum(B)}$. In X^m, *loc* and *port* are variables containing a location name and a port name respectively. In P^m, p^m designates the fresh port created for interacting with the monitor. Finally, $inst(T)$ is the set of transitions that should be instrumented: we instrument atomic components whose variables are needed by the monitor. T^m designates the transitions in the instrumented atomic component. We instrument the transitions in the corresponding atomic component that are modifying a variable involved with the monitor. If the state or the port of an atomic component is needed, all transitions are instrumented. For each transition $\tau \in inst(T)$, we add a fresh new transition to interact with the monitor. Transitions are also instrumented by adding new statements to record the state and the port name, if necessary.

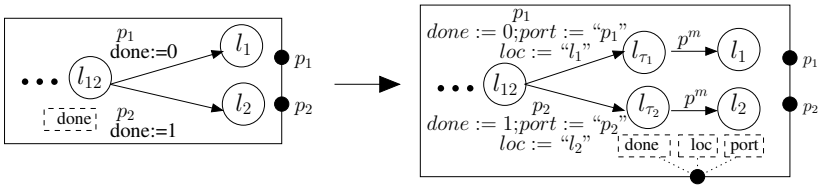


Fig. 6. Instrumentation of an atomic component

Example 8 (Instrumentation of an atomic component). Figure **??** illustrates the instrumentation of the atomic component depicted on the left-hand side into the instrumented component on the right-hand side. For instance, supposing that the state should be monitored, from the transition $\tau_1 = (l_0, p_1, \texttt{true}, f_{\tau_1}, l_1)$ with $f_{\tau_1} = [done := 0]$, we create a new state l_{τ_1} and the transitions $in(\tau_1) = (l_0, p_1, \texttt{true}, f_{in}, l_{\tau_1})$ with $f_{in} = [done := 0; loc := \text{“}l_1\text{”}; port := \text{“}p_1\text{”}]$, and $out(\tau_1) = (l_{\tau_1}, p_1, \texttt{true}, [\,], l_1)$.

5.3 Creating an Atomic Component from a Monitor

We present how an abstract monitor $\mathcal{A}$ is transformed into a BIP monitor $M^{\mathcal{A}}$ that mimics the behavior of $\mathcal{A}$. The generated BIP monitor receives events from the monitored system and processes them to produce the same verdicts as the initial abstract monitor. To do so, we expect monitors to adhere to two properties that will be used to ensure that the monitored system behave as the initial system.

Transformation of an abstract monitor into a BIP monitor. From an abstract monitor (cf. Definition **??**) given as an XML file, we construct the corresponding atomic component in BIP that interacts with the instrumented atomic components and produces verdicts following the behavior of the original monitor.

Definition 19 (Building monitors in BIP). *From a monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$, we define the corresponding atomic component $M^{\mathcal{A}} = (P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ as an atomic component implementing its behavior:*

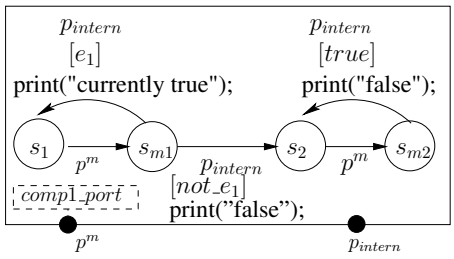
- $X = occur(\Sigma)$,
- $P = \{p^m[X], p_{intern}[\emptyset]\}$,
- $L = \Theta^{\mathcal{A}} \cup \{q_{mi} \mid q_i \in \Theta^{\mathcal{A}}\}$ *where each q_{mi} state is a fresh state associated to a q_i,*
- $T = T_1 \cup T_2$, *where*
  - $T_1 = \{(q_i, p^m, \texttt{true}, [\,], q_{mi}) \mid q_i \in \Theta^{\mathcal{A}}\}$,
  - $T_2 = \{(q_{mi}, p_{intern}, a, \texttt{print}(ver^{\mathcal{A}}(q'_i)), q'_i) \mid q_i \xrightarrow{a}_{\mathcal{A}} q'_i \wedge (q_i, p^m, \texttt{true}, [\,], q_{mi}) \in T_1\}$.

We note $M^{\mathcal{A}} = BuildMon(\mathcal{A})$ and call $M^{\mathcal{A}}$ a BIP monitor. T_1 denotes the set of transitions used to interact with the composite component. T_2 is the set of transitions used to display verdicts following the behavior of the original monitor $\mathcal{A}$. The set of variables of the monitor is the set of variables used in the specification (as in Section **??**).

Example 9 (Transforming an abstract monitor into a BIP monitor). Figure **??** illustrates the transformation of Definition **??**. The atomic component in Fig. **??** is transformed into the BIP monitor in Fig. **??**.

```
<VerificationMonitor>
    <State id="s1" initial="true">
        <Transition event="e1" nextState="s1" output="currently_true"/>
        <Transition event="not_e1" nextState="s2" output="false"/>
    </State>
    <State id="s2">
        <Transition event="true" nextState="s2" output="false"/>
    </State>
</VerificationMonitor>
```

(a) Abstract Monitor



(b) BIP Monitor

Fig. 7. Transforming an abstract monitor into a BIP Monitor.

The following corollary states that if the initial abstract monitor adheres to the readiness and determinism properties (see Definition **??**), then similar properties hold on the guards of the generated monitor.

Corollary 1 (Generating a BIP monitor preserves readiness and determinism). *Under the hypothesis that $\mathcal{A}$ adheres to readiness and determinism, the BIP monitor $M^{\mathcal{A}} = BuildMon(\mathcal{A})$ adheres to the readiness and determinism properties in the following sense. We transpose the notion of readiness and determinism defined for abstract monitors, for $l \in L$, let $guards(M^{\mathcal{A}}, l) = \{g_\tau \mid \tau = (l, \_, \_)\}$ be the set of guards of the transitions that can be fired in location l in the BIP monitor $M^{\mathcal{A}}$. We have:*

$$
\begin{array}{lll}
\forall l \in L, \forall v \in [X \to \text{Data}] : & & \\
\quad \bigvee_{g \in guards(M^{\mathcal{A}}, l)} g(v) & & \textit{(readiness)} \\
\wedge \quad \forall g_1, g_2 \in guards(M^{\mathcal{A}}, l) : g_1 \neq g_2 \Rightarrow \neg(g_1(v) \wedge g_2(v)) & & \textit{(determinism)}
\end{array}
$$

5.4 Connections

The next step of our transformation is to define the connectors between:

- $\pi(\Gamma(\{B_1^m, \ldots, B_n^m\}))$ the composite component consisting of instrumented atomic components where for $i \in [1, n]$ $B_i^m = \text{Instrum}(\text{B}_\text{i})$ (see Definition **??**), already connected with a set of connectors Γ with a priority model π, and
- the BIP monitor $M^{\mathcal{A}} = BuildMon(\mathcal{A})$ obtained from an abstract monitor $\mathcal{A}$, (see Definition **??**).

This is done by the following transformation that augments the existing set of connectors Γ and the priority model π.

Definition 20 (Connections). *Given $M^{\mathcal{A}}$ a BIP monitor and $\pi(\Gamma(\{B_1^m, \ldots, B_n^m\}))$ a composite component obtained as described above, the monitored composite component is $\pi^m(\Gamma^m(B_1^m, \ldots, B_n^m, M^{\mathcal{A}}))$, where,*

- $\Gamma^m = \Gamma \cup \{\gamma_1, \gamma_2\}$ *where* $\gamma_1 = (P_{\gamma_1}, t_{\gamma_1}, \texttt{true}, F_{\gamma_1})$, $\gamma_2 = (M^{\mathcal{A}}.p_{intern}, t_{\gamma_2}, \texttt{true}, \emptyset)$, *and,*
  - $P_{\gamma_1} = \{B_i.p^m[X_i^m]\}_{B_i \in comp(\Sigma)} \cup \{M^{\mathcal{A}}.p^m\}$;
  - $t_{\gamma_1} : P_{\gamma_1} \to \{\texttt{true}, \texttt{false}\}$, *where,* $\forall B_i \in comp(\Sigma)\ t_{\gamma_1}(B_i.p^m) = \texttt{true}$ *and* $t_{\gamma_1}(M^{\mathcal{A}}.p^m) = \texttt{false}$;

- *F_{γ_1}, the update function, is the identity data transfer from the variables in the ports of the interacting components B_i ($i \in [1, n]$) to the corresponding variables in the monitor port;*
- *the type of the port $M^{\mathcal{A}}.p_{intern}$ in the connector γ_2 is synchron ($t_{\gamma_2}(M^{\mathcal{A}}.p_{intern}) =$ `false`, that is one and only one interaction is defined by this connector: γ_2, see Definition* **??***);*

– $\pi^m = \pi \cup \{(a, a') \mid a \in \cup_{\gamma \in \Gamma} \mathcal{I}(\gamma) \wedge a' \in \mathcal{I}(\gamma_1) \cup \mathcal{I}(\gamma_2)\}$.

Connecting the instrumented atomic components and the BIP monitor consists in modifying the set of connectors and the priority model. Two connectors are added: γ_1, used by the monitor to retrieve the state of the system (e.g., the values of some variables), and, γ_2 (which is internal), used by the monitor to determine the verdict and move to a state where the monitor can receive further information from the monitored system. The priority model is augmented by giving more priority to the interactions defined by γ_1 and γ_2 than those defined by Γ (illustrated in Fig. **??**). Modifying the priority model ensures that, after execution of an interaction by the involved components, the monitor produces a verdict before involving other interactions.

5.5 Summary and Discussion

We end up this section by providing a summary, giving intuition about the correctness of our transformation and discussing some features about our framework. The complete proof is in Appendix **??**.

Summary. We propose a 4-stage approach to introduce runtime verification for CBS. Our method directly integrates an abstract monitor in a CBS. Thanks to the BIP framework, monitoring of a specification can be taken into account at design time. Moreover, the actual system, automatically generated from the augmented BIP model, is runtime-checked.

Some intuition about the correctness. The correctness proof is omitted for the sake of readability and can be found in Appendix **??**. We prefer here to give some intuition. The correctness relies on the following informal arguments. Our transformations do not modify the data nor the behavior induced by the initial interactions. No deadlock is introduced because the synthesized BIP monitor is always ready to receive events from the instrumented components. Finally, the priorities introduced when connecting the instrumented components to the BIP monitor (Section **??**) guarantee that the monitor always receives fresh data, i.e., the latest system state.

Remark 2 (About the initial state). In this section, we have not detailed how the monitor retrieves the initial state of the system. Actually, for each atomic component, the initial state is modified so as to add a transition labelled with the port p_m, synchronized with the monitor, so as to evaluate the initial state of the system. Our implementation, RV-BIP, instrument atomic components this way (see Section **??**).

Remark 3. In the transformation proposed in this section, one monitor is generated for the whole composite component. An adaptation of this framework could consist in generating several monitors according to the underlying architecture. Synchronization between the monitors would have to be defined.

Remark 4 (Genericity and connection with other runtime verification tools). The third stage of our transformation consists in transforming an existing abstract monitor into a BIP monitor mimicking its behavior. As we indicated before, the abstract monitor can be obtained from various existing tools dedicated to monitor synthesis. Another alternative, not developed here, is to use external monitor connected to the BIP monitor using the possibility of calling external functions in BIP. Two placements of the monitor are possible according to whether the monitor executes in the same memory space as the program (inline monitoring) or not (outline monitoring). Inline monitoring is possible with a tool such as R-MOR [**?**] where the BIP monitor directly queries a synthesized C monitor. Outline monitoring, using potentially any RV tool, remains also possible through some form of communication initiated by some C code called by the monitor.

6 Implementation and Evaluation

6.1 RV-BIP: A tool for Runtime Verification of BIP systems

RV-BIP is a Java implementation (~ 2500 LOC) of the transformations described in Section **??**, and, is part of the BIP distribution. RV-BIP takes as input a BIP system and an abstract monitor (an XML file) and then outputs a new BIP system whose behavior is monitored. RV-BIP uses the following modules (see Fig. **??**):

- *Extraction:* this module extracts the components and the corresponding variables used in the monitor. It takes as input an abstract monitor and then outputs a list of components with their corresponding variables,
- *Atomic Transformation:* this module instruments the atomic components selected from the extraction module. It takes as input the output of the *Extraction* module and a BIP file containing the original BIP system,
- *Building Monitor:* this module takes as input an abstract monitor and then outputs the corresponding atomic component,
- *Connections:* this module constructs the new composite component whose behavior is monitored. It takes as input the output from the *Atomic Transformation* and *Building Monitor* modules and then outputs a new composite component.

6.2 Case Study: a Robotic Application

We experimented RV-BIP on a robotic application modeled in BIP: Dala robot [**?**,**?**]. The Dala robot is a large and realistic interactive system. Dala is an infinite system (in terms of states and transitions) that cannot be directly model-checked.

The functional level of the Dala robot consists of a set of modules. A module is composed of a set of services corresponding to different tasks and a set of posters where the produced data is stored and exchanged between different modules. In this section, due to the lack of space, we present a simplified model of the modules with only the services related to two properties among those we runtime checked.

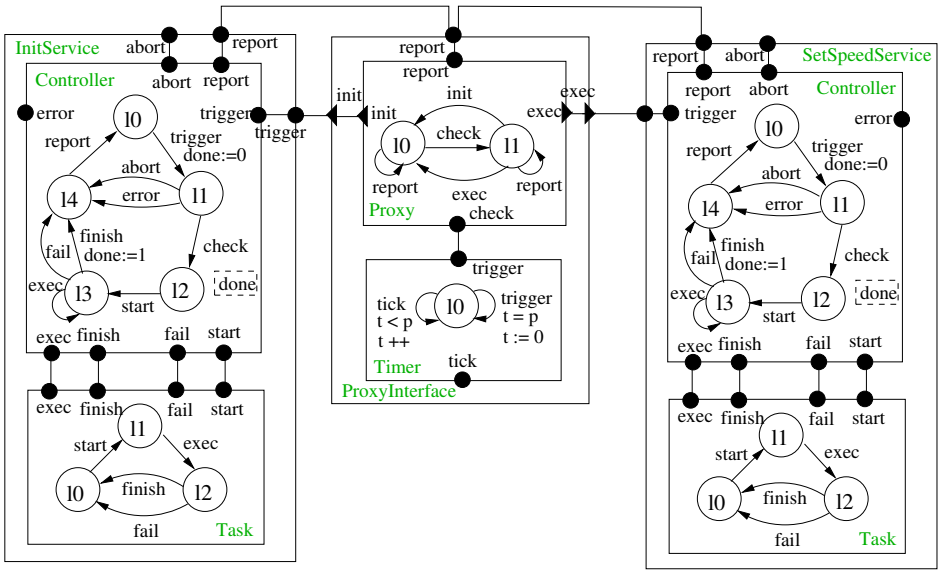


Fig. 8. Two services involving the ordering specification

Simple execution order. Figure **??** shows a simplified model of Dala. It consists of 3 components: *Proxy-Interface*, *InitService* and *SetSpeedService*. *ProxyInterface* communicates with the control layer using the mailbox by executing the transition *check*. *InitService* is responsible for the initialization of the module and *SetSpeedService* performs the main task of the module. According to the received request, *Proxy* triggers either *InitService* or *SetSpeedService*. Each service has a status variable *done*: value 1 means that the corresponding task has been successfully executed. A service can be triggered through the port *trigger*, then it executes its task by taking the transition *start* and finally it returns to the initial location by the transition *finish* when the task is done. The execution order of some services is important. In this module, *InitService* initializes the robot and should be successfully executed before *SetSpeedService* sets the speed parameter of the robot. This requirement is formalized as "φ_1 and φ_2", see Table **??**.

Data freshness. In Dala, the modules communicate by a set of posters. Data generated by a module is written in a poster that can be accessed by another module. The behavior of the robot might depend on this data, therefore it is necessary that the data is up to date: the data read by a service of a module (called $Reader$) must be fresh enough compared to the moment it has been written (by a service called $Writer$). If t_1 and t_2 respectively are the moments of reading and writing actions, then the difference between t_2 and t_1 must be less than a specific duration δ, i.e., $(t_2 - t_1) \leq \delta$. In the model, the time counter is implemented by a component $Clock$, and the $tick$ transition occurs every second. This requirement is formalized as "φ_3 and φ_4", see Table **??**.

Mutual exclusion. The services in robot Dala share the same set of posters. Different services must not access and modify data in a poster at the same time. This is an important and critical property for the robot to function correctly. We enforce this property by adding a constraint: a poster allows a writer to trigger its writing process only if the poster is not occupied by any other writer. A variable *concurrent* is used to

Table 1. Formalization of the requirements for the Dala robot

φ_1 : $(e_1)^*$, where, e_1 : $(SetSpeedService.port == trigger \wedge ProxyInterface.port == exec)$ $\Rightarrow (InitService.done == 1)$
φ_2 : $(e_1 \cdot e_2)*$, where, e_1 : $InitService.port == finish$ e_2 : $SetSpeedService.port == trigger$
φ_3 : $(e_1)^*$, where, e_1 : $(Reader.port == read \wedge poster.port == read \wedge Clock.port == getTime)$ $\Rightarrow (Clock.time - poster.wrtime \leq 2)$
φ_4 : $(e_1 \cdot (\epsilon + e_2 + e_2 \cdot e_2) \cdot e_3)^*$, where, e_1 : $Writer.port == write$ e_2 : $Clock.port == tick$ e_3 : $Reader.read == read$
φ_5 : $(e_1)^*$, where, e_1 : $(poster.concurrent \leq 1)$
φ_6 : $\Big[\big[(\neg e_2 + \neg e_3)^*.e_1.(\neg e_2 + \neg e_3)^*\big].$ $\big[(\neg e_1 + \neg e_3)^*.e_2.(\neg e_1 + \neg e_3)^*\big].$ $\big[(\neg e_1 + \neg e_2)^*.e_3.(\neg e_1 + \neg e_2)^*\big]\Big]^*$, where, e_1 : $(Writer_1.port == write \wedge poster.port == write \wedge clck.port == getTime)$ e_2 : $(Writer_2.port == write \wedge poster.port == write \wedge clck.port == getTime)$ e_3 : $(Writer_3.port == write \wedge poster.port == write \wedge clck.port == getTime)$

represent the number of writers that are accessing a poster. This variable is increased (or decreased) by one when a writer starts (or finishes respectively) its writing process. The property is then checked by using this variable: a write can use a poster only if the value of *concurrent* of the poster is 0 meaning that no other writer is using the poster. This requirement is formalized as "φ_5", see Table **??**.

Complex execution order. A more complex property on the execution order involves several writers: they periodically write data to posters in a specific order. We considered this property on 3 writers: $Writer_1$, $Writer_2$ and $Writer_3$. The writing order in every period must always be as follows: $Writer_1$ writes to a poster first, then $Writer_2$ can write only when $Writer_1$ finishes, $Writer_3$ can write only when $Writer_2$ finishes, and the same for the next periods. To do so, each writer is assigned a unique id that is passed to the poster when it starts using the poster. This id is then used to determine the last writer that used the poster. For example, when $Writer_2$ wants to access a poster, it has to check whether the id stored in the poster corresponds to $Writer_1$ or not. This requirement is formalized as "φ_6", see Table **??**.

Experiments. Table **??** reports results on checking the ordering and freshness properties of the Dala robot. *Ordering violated* and *Ordering guaranteed* correspond to the model presented in Fig. **??** where the first one might have the violation of the ordering specification whereas the second one always guarantees it. Likewise, *Data freshness violated* and *Data freshness guaranteed* correspond to the model presented in Fig. **??** where

Table 2. Results of monitoring the requirements Execution order and Data freshness

	time-no-monitor	**specification**	**optimized**		**not-optimized**	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering violated	1.896	φ_1	2.045	7.8	9.163	383.0
		φ_2	1.953	3.0	9.192	384.0
Ordering guaranteed	1.836	φ_1	1.984	8.0	8.900	384.0
		φ_2	1.889	2.8	8.896	384.0
Data freshness violated	1.638	φ_3	1.684	2.8	4.337	164.0
		φ_4	1.682	2.6	3.773	130.0
Data freshness guaranteed	1.634	φ_3	1.678	2.6	4.383	168.0
		φ_4	1.690	3.4	3.782	131.0
Complex ordering violated	5.359	φ_5	5.555	3.66	6.410	19.6
Complex ordering guaranteed	7.057	φ_5	7.405	4.9	8.415	19.2
Mutual exclusion violated	5.299	φ_6	5.540	4.5	6.402	20.81
Mutual exclusion guaranteed	7.024	φ_6	7.366	4.86	8.405	19.66

the first one might have the violation of the freshness specification whereas the second always guarantees it. In Table **??**, the columns have the following meanings:

- the column *time-no-monitor* indicates the execution time without monitoring, the column *specification* shows the monitored specification
- the column *optimized* reports the execution time and the overhead obtained with the monitor that interacts only with the two components involved in the specification, and
- the column *not-optimized* reports the execution time and the overhead obtained with a monitor that observes all components of the system (even the ones that are not involved in the specification).

The results substantiate our claim that if we monitor only components involved in the specification, using the abstraction technique defined in Section **??** and implemented in Section **??**, the overhead is reduced significantly.

7 Related Work

We propose to overview and compare work related to the approach proposed in this paper. We distinguish three kinds of related approaches:

- static verification techniques (e.g., model-checking, static analysis) dedicated to component-based systems (Section **??**);
- general-purpose runtime verification approaches dedicated to monolithic programs (Section **??**);
- runtime verification approaches dedicated to component-based systems (Section **??**).

7.1 Static/Design-time Verification of Component-Based Systems

With the growing demand of scalability and complexity for systems, it is even more important to use verification techniques to determine whether a designed system meets its requirements. Static formal verifica-

tion [?,?,?] is based on mathematical techniques to prove or disprove the correctness of a design w.r.t. a given formal specification. These techniques search for input patterns which lead to violations of the desired properties and prove the correctness when such violations do not exist. Existing formal verification methods for component-based systems are based on either static analysis or on model-checking.

Approaches based on static analysis consist in computing specific invariants in order to abstract the state space. Although approaches based on static analysis are less sensitive to state explosion, they still suffer from some limitations. First these techniques are rather limited in terms of the properties they can check: they are mostly limited to safety properties and thus some interesting behavioral properties remain out of the scope of these techniques. Moreover, since these approaches rely on abstraction and over approximation of the state space, they usually yield several false positives.

Model-checking is based on an exhaustive exploration of the state space of the model obtained from the operational semantics of the specification language. For large systems, this exploration leads to a very large number of states (the well-known state explosion problem). Despite recent advances in model-checking, the state-explosion problem is far from being solved and limit the use of these methods in component-based systems where the state space tends to become huge due to the number of possible configurations and interactions between components. Moreover, techniques based on compositional verification [?,?,?] (less sensitive to state explosion) require to over-approximate the behavior of the unknown parts of the system - as it can be the case in BIP when using external C functions.

A compositional verification method based on invariants for checking safety properties in component-based systems is proposed in [?,?]. The method over-approximates the set of reachable states using both local invariants that characterize local constraints of atomic components and global constraints that are induced by strong synchronization between components. Although the method has been successfully applied to large-scale and complex systems, the use of invariants can deal only with safety properties and might produce many false positive counter examples.

Another compositional approach is design-by-contract [?,?] that considers a property provided by a component as a contract between this component and its environment. A contract is expressed as a pair consisting of an assumption (that the environment must satisfy) and a guarantee (the property satisfied by each component). For instance [?] provides a method that searches an implementation model that satisfies a given contract. Although the experimental results are promising, it is not always possible to find an implementation model that satisfies a given property. Moreover, the composition of contracts in concurrent systems can be very expensive.

The limitations of static validation techniques led us to investigate the use of *runtime verification* as an *alternative and complementary* technique to validate CBS.

7.2 Runtime Verification of Monolithic Programs (with mathematically-proven guarantees)

Contrarily to static-verification techniques which are exhaustive, runtime verification techniques generally focus on a single execution of the system under scrutiny. While static verification techniques can produce false-positives, runtime verification techniques can miss some property violations. However, runtime verification techniques are complementary since they apply to deployed systems.

Over more than a decade, the field of *runtime verification* has produced many frameworks dedicated to the verification of the behavior of monolithic programs w.r.t. user-defined specifications. Many tools have been proposed as implementations of runtime verification frameworks. One of the most successful frameworks is Java-MOP (see [**?**] for an overview). Java-MOP can use input specifications written in many formalisms (e.g., LTL, regular expressions, context-free grammars). Java-MOP generates an AspectJ aspect that instruments the underlying program (using weaving) and embeds the (automatically generated) monitor. Besides its generality, Java-MOP is also efficient as demonstrated by experimentation. A series of tools and approaches are based on the (less efficient) paradigm of rewriting and focus on expressiveness of the specification formalism. Some of the main efforts are Eagle [**?**], RuleR [**?**,**?**], LogScope [**?**], and TraceContract [**?**]. Eagle handles LTL formulae and uses *progression* [**?**]. RuleR is a more general system where specifications are encoded as a set of rewrite rules. This confers RuleR the ability to handle very expressive specifications. From an abstract point of view, LogScope is a variant of RuleR internally using state-machines. TraceContract is an embedding of LogScope in the Scala programming language (as an internal domain-specific language).

Other efforts include TraceMatches [**?**], JLO [**?**], and LARVA [**?**,**?**]. TraceMatches extends AspectJ by allowing to write regular expressions over pointcuts. JLO generates monitors from LTL formulae where events are AspectJ pointcuts. Finally, LARVA monitors different specification formalisms such as Lustre and duration calculus. LARVA translates specifications into the so called dynamic event timed automata and then uses AspectJ to weave the monitor.

Runtime verification frameworks with mathematically-proven guarantees. Several runtime verification frameworks provide mathematical guarantees on their specifics. Most of the frameworks prove the correctness of their monitor-synthesis algorithms, i.e., the verdict produced by the monitor on any trace follows the semantics of the specification used to generate the monitor. Similarly, Rosu et al. [**?**] recently provided proofs of the mathematical correctness of their algorithms for monitoring parametric specifications. Monitoring parametric specifications involves additional algorithms to handle the data carried out in events. A different kind of guarantee is provided by Barringer et al. in RuleR [**?**]: they proved the correctness of a translation from LTL to RuleR specifications.

Comparison with our approach. There are several noteworthy differences between our approach and existing runtime verification techniques.

Our approach differs mainly because we do not target monolithic programs but component-based systems. Moreover, related approaches rely on aspect-oriented programming (AOP). For instance, Java-MOP automatically generates the needed AspectJ aspect while RuleR expects the user to write an aspect. In all cases, existing runtime verification frameworks use AOP to instrument the system (i.e., inserting code to observe relevant events). Since the technology of AOP is not available for component-based systems, we define our own instrumentation for BIP systems. In some sense, the proposed instrumentation mimics the usual workflow adopted by most runtime verification frameworks. Indeed, we extract from the specification the relevant events and variables that need to be observed and we directly add instrumentation code at rele-

vant places in the system. Dealing with BIP system allows us to define a formal definition of the performed instrumentation, contrarily to other frameworks using AspectJ.[10]

Compared to the existing approaches providing mathematical guarantees about their correctness, we provide mathematically proven guarantees on the correctness of our instrumentation technique. In our approach, the monitor is considered to be an input. Previously described approaches remain compatible and are complementary.

7.3 Runtime Verification of Component-Based Systems

Specification and runtime verification of the behavior of CBS have received less research attention. A first series of approaches specify the behavior of components in terms of pre and post-conditions (e.g., with JML) or assertions (e.g., using Eiffel). More recently and closer to our work is the LIME specification language [**?**] that allows runtime monitoring of temporal properties for component interfaces. Components are black boxes and LIME specifications define how components should interact with an external application by describing a desired behavior on the calls and returns over the interface. Concurrently and independently, Dormoy et al. proposed an approach to runtime check the correct reconfiguration of components at runtime [**?**]. They propose to check configurations over a variant of RV-LTL where the usual notion of state is replaced by the notion of component configuration. RV-LTL is a 4-valued variant of LTL dedicated to runtime verification introduced in [**?**] and used in [**?**].

Comparison with our approach Compared to previous dynamic techniques, our approach offers several advantages. First, we use the latest advances in runtime verification using an expressive 4-valued truth-domain allowing our monitor to be generated using any monitor synthesis framework. Our RV framework only uses information about the events used in the specification. Even though the monitors presented in this paper are presented as regular properties[11], the expressiveness of the BIP language confers our monitors a potential to be Turing-complete. For instance, adding internal variables to the monitor can be done with no particular difficulty. Moreover, compared to [**?**], our approach is not limited to monitoring component interfaces. It is often the case that components come with an abstract behavioral model, i.e., components are gray boxes instead of black boxes. Our monitoring framework supports the three kinds of approaches (black, grey, and white). Furthermore, the specifications considered for BIP systems use locations spanning over several components allowing the specification of global behaviors of the system in composition. Our approach offers several advantages compared to Dormoy et al. [**?**]. First, our approach is not bound to temporal logic since it only requires a monitor written as a finite-state machine. This state-machine can be then generated by several already existing tools (e.g., Java-MOP) since it uses a generic format to express monitors. Thus, existing monitor synthesis algorithms from various specification formalisms can be re-used, up to a syntactic adaptation layer. Second, the instrumentation of the initial system and the addition of the monitor is formally defined, contrarily to [**?**] where the process is only overviewed. Moreover, the whole

[10] There are some approaches proposing a formal semantics of aspect-oriented programming, but these approaches work mainly on abstract models of the underlying programming language. Moreover, to the best of our knowledge, no RV framework has proposed a formalization of its instrumentation process.

[11] Because we use as input a monitor specified as a finite-state machine.

approach leverages the formal semantics of BIP allowing us to provide a formal proof of the correctness of the proposed approach. All these features confers to our approach a higher-level of confidence.

8 Conclusion and Future Work

Conclusion. This paper introduces runtime verification as a complementary validation technique for component-based systems written in the BIP framework. Our technique is based on a general and expressive runtime verification framework with a 4-valued truth-domain. Our solution dynamically builds a minimal abstraction of the current runtime state of the system so as to lower the overhead. Generating monitors directly as BIP components confers to our approach several advantages. First, thanks to the C code generator of BIP, we are able to generate actual monitored C programs that can be directly deployed. Second, our approach remains compatible with previously proposed runtime verification frameworks in two respects. First, thanks to the generic format of the input abstract monitor, a light adaptation layer is needed to adapt monitors generated by other existing tools. Second, using the possibility of calling external C code, our BIP monitors can use external monitors as a service to evaluate the state of the current system. Another advantage is that our approach can adapt to different level of modelling since it does not make any particular hypothesis except the ability to observe the BIP events involved in the specification. A last advantage is that, by targeting BIP systems, we can reuse several tools and research insights proposed by previous endeavors on this topic (e.g., [**?**,**?**,**?**]). Our approach has been implemented in RV-BIP that smoothly integrate in the existing BIP tool-set. Finally, experimental evaluations on a robotic application substantiate our claims about the effectiveness of our instrumentation and the feasibility of our approach.

Some perspectives. Our aim with this article was to propose a first formal approach to runtime verification of component-based systems. Runtime verification as a field has contributed to checking the correctness of object-oriented programs. We believe that this first approach can serve as a headway towards transferring the lessons learned and the frameworks developed to the new challenges that component-based systems require. More specifically, we propose some research perspectives. A first direction is to combine the recent advances in RV that use static analysis (see e.g., [**?**]). In RV, using static analysis techniques may reduce the overhead induced by a monitor by disabling unnecessary runtime checks. Also related to overhead reduction, a dynamic instrumentation technique [**?**], enabling the monitor to remove connectors when they are not needed anymore, would reduce the overhead even more. Another possible direction is to extend the proposed framework for runtime enforcement [**?**]. Runtime enforcement is an extension of RV aiming at circumventing property violation and provides better confidence in system behaviors. A more practical direction is to connect RV-BIP to the various existing monitor synthesis tools available within the RV community. Finally, given the recent advances in the multi-core and the Network on Chip technologies, we plan to customize our transformations for generating distributed monitors rather than a centralized monitor. Then, using the techniques presented in [**?**,**?**], we plan to automatically generate correct and efficient distributed implementations running on distributed platforms.

Acknowledgments. The authors would like to warmly thank the anonymous reviewers for their insightful remarks.

A A Proof of Correctness of the Proposed Approach

In order to prove the correctness of our approach, we proceed according to the following stages:

1. Introducing a suitable abstraction of the system. In this abstraction, some data is discarded to focus only on the behavior of the system (Section **??**).
2. Introducing some intermediate definitions and lemmas (Section **??**).
3. Proving that the initial system and the instrumented system are observationally equivalent by showing a weak bi-simulation between them. This is the cornerstone of the correctness of our approach in the sense that it demonstrates that our transformation preserves the initial behavior of the system up to some actions of the monitor. This result is proved in Section **??**.
4. Proving that our transformation correctly transforms the initial system (Section **??**), using some intermediate lemmas from previous stages.

In the following proofs, we will consider several mathematical objects in order to prove the correctness of our framework:

- an abstract monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$;
- a BIP monitor $M^{\mathcal{A}} = (P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ generated form $\mathcal{A}$, i.e., $M^{\mathcal{A}} = BuildMon(\mathcal{A})$;
- a composite component $B = \pi(\Gamma(\{B_i\}_{i \in [1,n]}))$ along with its behavior $C = (Q, A, \longrightarrow)$;
- the instrumented composite component $B^m = \pi^m(\Gamma^m(\{B_i^m\}_{i \in [1,n]} \cup \{M^A\}))$ along with its behavior $C^m = (Q^m, A^m, \longrightarrow_m)$. B^m is obtained from B by following the procedure described in Section **??**.

A.1 Abstracting Data

With the objective of simplifying the following proofs, we introduce an abstraction consisting in analyzing the behavior of the involved components without considering some of the data. This abstraction is possible as one can notice that our transformations modify the values of some newly introduced variables but preserve the values of the variables that were present in the initial system.

Recall that a state of an atomic component is defined as a 3-tuple $q = (l, v, p)$ where $l \in L$ is the control state, $v \in [X \to \text{Data}]$ is a valuation of the variables X of the atomic component, $p \in P$ is the port labelling the last executed transition. To simplify proofs, we introduce an abstraction that consists in omitting the variables defined in the original atomic components. This abstraction is obtained by discarding some functions and guards defined in the connectors and transitions. Moreover, a state of an atomic component $q = (l, v, p)$ for some $l \in L$, $v \in [X \to \text{Data}])$ and $p \in P$ reduces to the actual control state l in the abstracted semantics. Consequently, a (global) state of B is a tuple consisting of the local states of its constituent atomic components. That is, the behavior C of the composite component $B = \Gamma(\{B_1, \ldots, B_n\})$

is a transition system $(Q, \gamma, \longrightarrow)$, where $Q = Q_1 \times \ldots \times Q_n$ (with $\forall i \in [1, n] : Q_i = B_i.L$) and $\longrightarrow$ is the least set of transitions satisfying the rule:

$$\frac{\begin{array}{c}\exists \gamma \in \Gamma : \gamma = (P_\gamma, t) \qquad \exists a \in \mathcal{I}(\gamma) \wedge a = \{p_i\}_{i \in I} \wedge I \subseteq [1, n] \\ \forall i \in I : \; q_i \xrightarrow{p_i}_i q_i' \qquad \forall i \notin I : \; q_i = q_i'\end{array}}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

Note that since data is abstracted, an interaction γ now consists of a set of ports $\mathcal{P}_\gamma$ and the function t specifying the types of ports. The notion of execution (run) of composite components, in this abstracted semantics, transposes easily from Definition **??** to abstract behaviors. Moreover, in the following, to lighten notation, given a state $q \in Q$ we do not make the distinction between $[\![q]\!]$ and q.

A.2 Preliminary Definitions and Lemmas

We recall and introduce some definitions and intermediate results on our transformations that will be used when proving our central result in Section **??**.

Observational equivalence and bi-simulation. Let us recall the notion of *observational equivalence* of two transition systems. It is based on the usual definition of weak bisimilarity [**?**], where β and β-transitions are considered unobservable.

Definition 21 (Weak simulation). *Given two transition systems $S_1 = (Q_1, P_1 \cup \{\beta\}, \longrightarrow_1)$ and $S_2 = (Q_2, P_2 \cup \{\beta\}, \longrightarrow_2)$, the system S_1* weakly simulates *the system S_2, if there is a relation $R \subseteq Q_1 \times Q_2$ such that the two following conditions hold:*

1. $\forall (q, r) \in R, \forall a \in P : \; q \xrightarrow{a}_A q' \implies \exists r' : \; (q', r') \in R \wedge r \overset{\beta^* \cdot a \cdot \beta^*}{\longrightarrow}_B r'$, *and*
2. $\forall (q, r) \in R : \; q \xrightarrow{\beta}_A q' \implies \exists r' : \; (q', r') \in R \wedge r \xrightarrow{\beta^*}_B r'$

Equation *1.* says that if a state q simulates a state r and if it is possible to perform a from q to end in a state q', then there exists a state r' simulated by q' such that it is possible to go from r to r' by performing some unobservable actions, the action a, and then some unobservable actions. Equation *2.* says that if a state q simulates a state r and it is possible to perform an unobservable action from q to reach a state q', then it is possible to reach a state r' by a sequence of unobservable actions such that q' simulates r'.
In that case, we say that the relation R is a weak simulation over S_1 and S_2 or equivalently that the states of S_1 are similar to the states of S_2. Similarly, a weak bi-simulation over S_1 and S_2 is a relation R such that R and $R^{-1} \overset{\text{def}}{=} \{(q_2, q_1) \mid (q_1, q_2) \in R\}$ are both weak simulations. In this latter case, we say that S_1 and S_2 are *observationally equivalent* and we write $S_1 \sim S_2$.

System stability. We define now a notion of system stability. Intuitively, a system will be unstable when the system has sent some event to the monitor and the monitor is currently processing this event. Below, we exhibit some properties of our transformed system related to stability.

Following Definition **??**, the set A^m of interactions of B^m can be partitioned into (1) the set A of initial interactions (present in the initial composite component), (2) the set $A^1 = \mathcal{I}(B^m.\gamma_1)$ of interactions used by

the monitor to observe the behavior of the system, and (3) the set $A^2 = \mathcal{I}(B^m.\gamma_2)$ of internal interactions of the monitor to move to the next state. We have $A^m = A \cup A^1 \cup A^2$. Moreover, the pairwise intersection of A, A^1, A^2 is empty. Observational equivalence considers that all interactions in $A^1 \cup A^2$ are labelled by unobservable events, denoted by β.

Definition 22 (Stable). *Given a state $q^m = (q_1^m, \ldots, q_n^m, q_{mon}) \in Q^m$, the predicate $is\_stable \in [Q^m \to \{\texttt{true}, \texttt{false}\}]$ is defined as follows:*

$$is\_stable(q^m) \quad \textit{iff} \quad \forall i \in [1, n] : q_i^m \in B_i.L.$$

A state of a composite component, consisting of an n-tuple of the state of some atomic components, is stable if each of the n states of the atomic component belongs to the uninstrumented system. That is, the constituting local states were not introduced by the transformation proposed in Definition **??**.

We now introduce the notion of *state stabilization*. Stabilizing a state consists in either doing nothing if this state is already stable or returning the next stable state reached by the system.

Definition 23 (State stabilization). *Let $q^m = (q_1^m, \ldots, q_n^m, q_{mon}) \in Q^m$ be a state, the function $stable : Q^m \to Q$ is defined as follows: $stable(q^m) = q$, where $q = (stable_1(q_1^m), \ldots, stable_n(q_n^m))$, where the intermediate functions $stable_i \in [B_i^m.L \to B_i.L]$, for $i \in [1, n]$, are defined as follows:*

$$stable_i(q_i^m) = \begin{cases} q_i^m & \textit{if } q_i^m \in B_i.L \\ q' & \textit{otherwise, where } \exists q : (q, p_m, q') \in B_i^m.T \end{cases}$$

Intermediate lemmas. We now propose some intermediate results characterizing the status of the global system w.r.t. the notion of stable states and stabilization. The first lemma is a direct consequence of the definition of the predicate is_stable and the notion of stabilization.

Lemma 1. *For a given state $q^m = (q_1^m, \ldots, q_n^m, q_{mon})$, we have $is\_stable(q^m) \Leftrightarrow stable(q^m) = (q_1^m, \ldots, q_n^m)$.*

The following lemma states that when the system is in an unstable state, i.e., some constituting atomic components have performed an instrumented transition, then the arriving state is such that the monitor can perform a transition labelled by p_m (and thus receive an environment from the components).

Lemma 2 (When the system is not stable the monitor waits for the system). *For every state $q^m = (q_1^m, \ldots, q_n^m, q_{mon}) \in Q^m$, the following property holds*

$$\neg is\_stable(q^m) \Rightarrow q_{mon} \xrightarrow{p^m}_{M^{\mathcal{A}}}$$

*where $\longrightarrow_{M^{\mathcal{A}}}$ is the transition relation of the monitor and p^m is the port used by components to communicate with the monitor (see Definition **??**).*

Proof. We distinguish two cases according to whether q^m is the initial state of the system or not. First, if q^m is the initial state of the system, then from Definition **??** we have $q_{mon} \xrightarrow{p^m}_{M^{\mathcal{A}}}$. Second, if q^m is not the

initial state of the system, let $q'^m = (q'^m_1, \ldots, q'^m_n, q'_{mon})$ be its predecessor and a be the interaction leading to q^m, that is $q'^m \xrightarrow{a}_m q^m$. The interaction a belongs either to A, A^1, or A^2 (where $\{A, A_1, A_2\}$ is the partition of the interactions of the instrumented components as defined in the paragraph *system stability*):

- If $a \in A$, then the state of the monitor at state q'^m is equal to the state of the monitor at state q^m. Indeed, the interactions in $\mathcal{I}(B^m.\gamma)$ consist only of the ports of the atomic components $\{B_i \mid i \in [1, n]\}$. Since the interaction defined by $\mathcal{I}(B^m.\gamma_2)$ has more priority than the interactions in $\mathcal{I}(B^m.\gamma)$, then necessarily in the current local state q_{mon} of the monitor, it is not possible to fire a transition with p_{intern} (i.e., $q_{mon} \overset{M^{\mathcal{A}}.p_{intern}}{\not\longrightarrow}_{M^{\mathcal{A}}}$). Otherwise the interaction $\{p_{intern}\}$ would be executed since the interaction defined by $\mathcal{I}(B^m.\gamma_2)$ consists only of the port $M^A.p_{intern}$, such an interaction has more priority than any other existing interaction in the system, and such an interaction would be enabled because of readiness.
- If $a \in A^1$, then $a \subseteq \bigcup_{i=1}^{n} \{B_i^m.p^m\}$. Using Definition **??** with maximal progress (Definition **??**) ensures that from the local states q_i^m, the port p^m is not enabled for all $i \in [1, n]$. Hence, we have $\forall i \in [1, n] : q_i^m \in B_i.L$, that is, $is\_stable(q^m)$.
- If $a \in A^2$, then $q'_{mon} \xrightarrow{p_{intern}}_{M^{\mathcal{A}}}$. Thus, the fact that $q_{mon} \xrightarrow{p^m}_{M^{\mathcal{A}}}$ follows directly from Definition **??**.

Lemma 3 (After an unstable state the system stabilizes). *Given a run $q^0 \cdot q^1 \cdots q^s$ of B^m such that $q^i \xrightarrow{a_i}_m q^{i+1}$ holds for all $i \in [0, s-1]$, we have:*

$$\forall i \in [0, s-1] : \neg is\_stable(q^i) \Rightarrow is\_stable(q^{i+1})$$

Proof. Let us consider $q^i = (q_1^i, \ldots, q_n^i, q_{mon})$ a non stable state (i.e., $\neg is\_stable(q^i)$) of the run with $i \in [0, s-1]$ (hence q^i is not the last state[12]). Let $q^{i+1} = (q_1^{i+1}, \ldots, q_n^{i+1}, q'_{mon})$ be the successor state of q^i in the run. Lemma **??** guarantees that the monitor is able to perform a transition labelled by p_m in q^i, that is $q_{mon} \xrightarrow{p^m}_{M^{\mathcal{A}}}$. Let us consider $Q_u = \{q_j^i \mid q_j^i \notin B_j.L\}$ be the set of locally unstable states. As q^i is not stable, Q_u is not empty. The set of possible interactions is the set of subsets of $\{B_j^m.p^m \mid q_j^i \in Q_u\} \cup \{M^{\mathcal{A}}.p^m\}$. Indeed, observe that first, these interactions have more priority than the interactions in $\mathcal{I}(B^m.\gamma)$, and second that the monitor is ready $q_{mon} \xrightarrow{p^m}_{M^{\mathcal{A}}}$ (it is not possible to execute any interaction in $\mathcal{I}(B^m.\gamma_2)$). Moreover, maximal progress (Definition **??**) guarantees that the executed interaction is $\{B_j^m.p^m \mid q_j^m \in Q_u\} \cup \{M^{\mathcal{A}}.p^m\}$. In turn, Definition **??** ensures that from all local states q_j^{i+1}, $j \in [1, n]$, the port p^m is not enabled. Thus, we have $is\_stable(q^{i+1})$.

A.3 Observational Equivalence between the Original and Transformed BIP Models

We are now ready to state and prove our central result.

Proposition 1. *The non-instrumented system is bi-similar to the instrumented system where interactions with the monitor and internal interactions of the monitor are considered to be unobservable actions, that is: $B^m \sim B$.*

[12] Otherwise the lemma holds vacuously.

Proof. Following Section **??**, we need to exhibit a relation R between the set of states Q^m of B^m and the set of states Q of B. We define $R \stackrel{\text{def}}{=} \{(q^m, q) \mid q^m \in B^m \wedge stable(q^m) = q\}$. We shall prove the three next assertions to establish that R is a weak bi-simulation:

(i) $\forall (q^m, q) \in R : q^m \xrightarrow{\beta}_m r^m \Longrightarrow (r^m, q) \in R$.
(ii) $\forall (q^m, q) \in R : q^m \xrightarrow{a}_m r^m \Longrightarrow \exists r \in Q : q \xrightarrow{a} r \wedge (r^m, r) \in R$.
(iii) $\forall (q^m, q) \in R : q \xrightarrow{a} r \Rightarrow \exists r^m \in Q^m : q^m \xrightarrow{\beta^* a}_m r^m \wedge (r^m, r) \in R$.

Proof of (i):

Let us suppose that $q^m \xrightarrow{\beta}_m r^m$, we have two cases according to the partition of interactions proposed in Section **??**:

- Case $\beta \in A^1$. Then $\beta \subseteq \bigcup_{i=1}^n \{B_i^m.p^m\}$. Let $q^m = (q_1^m, \ldots, q_n^m, q_{mon})$ and $r^m = (r_1^m, \ldots, r_n^m, r_{mon})$. Because $(q^m, q) \in R$, we have $q = stable(q^m) = (stable_1(q_1^m), \ldots, stable_n(q_n^m))$. We distinguish two sub-cases according to whether q_i^m is stable or not.
  - Let us suppose that q_i^m is a stable state, then we have $stable_i(q_i^m) = q_i^m$. From the local state q_i^m of the atomic component B_i, port $B_i^m.p^m$ is not enabled, hence after executing an interaction consisting only of ports p^m the local state q_i^m does not change, that is $q_i^m = r_i^m$ and $stable_i(r_i^m) = stable_i(q_i^m)$.
  - Let us suppose q_i^m is not a stable state, then $\exists q' \in Q_i^m : stable_i(q_i^m) = q' \neq q_i^m$. From the local state q_i^m, the port $B_i^m.p^m$ is enabled. Moreover, after executing the interaction β, the local state q_i^m becomes r_i^m, where $r_i^m = stable_i(q_i^m) = q'$ (because of maximal progress, see Definition **??**), and $r_i^m \in B_i.L$ (see Definition **??**), that is $stable_i(r_i^m) = r_i^m = stable_i(q_i^m)$. Therefore, $stable(r^m) = (stable_1(q_1^m), \ldots, stable_n(q_n^m)) = stable(q^m) = q$, thus $(r^m, q) \in R$.
- Case $\beta \in A^2$, that is $\beta = \{M^A.p_{intern}\}$. Hence, after executing β none of the local states q_i^m for $i \in [1, n]$ change (that is, $r_i^m = q_i^m$ for $i \in [1, n]$). Therefore, $stable(r^m) = (stable_1(q_1^m), \ldots, stable_n(q_n^m)) = stable(q^m) = q$, thus $(r^m, q) \in R$.

Proof of (ii):

Suppose that $q^m \xrightarrow{a}_m r^m$. Then $stable(q^m) = q^m$, that is, $is\_stable(q^m)$. Let $q^m = (q_1^m, \ldots, q_n^m, q_{mon})$ and $q = (q_1^m, \ldots, q_n^m)$, from state q interaction a is possible. Let r be the next state after executing a, that is $q \xrightarrow{a} r$. We distinguish two cases according to whether r^m is stable or not:

- If $is\_stable(r^m)$, then $r = (r_1^m, \ldots, r_n^m)$ where $r^m = (r_1^m, \ldots, r_n^m, r_{mon})$ (Definition **??**). Hence, $stable(r^m) = (stable_1(r_1^m), \ldots, stable_n(r_n^m)) = (r_1^m, \ldots, r_n^m) = r$, that is $(r^m, r) \in R$.
- If $\neg is\_stable(r^m)$, let s^m be the next state in the run after r^m, that is $r^m \xrightarrow{\beta}_m s^m$. Lemma **??** ensures that s^m is stable ($is\_stable(s^m)$), hence the interaction β is such that $\beta \subseteq \cup_{i=1}^n \{B_i^m.p^m\}$. As s^m is stable, then $stable(s^m) = (s_1^m, \ldots, s_n^m)$ (Lemma **??**), where $s^m = (s_1^m, \ldots, s_n^m, s_{mon})$. Moreover, since $\beta \subseteq \cup_{i=1}^n \{B_i^m.p^m\}$, then $stable(r^m) = (s_1^m, \ldots, s_n^m)$. Definition **??** ensures that $r = (s_1^m, \ldots, s_n^m)$. That is, $stable(r^m) = r$, thus $(r^m, r) \in R$.

Proof of (iii):

Suppose that $q \xrightarrow{a} r$. Let $q^m = (q_1^m, \ldots, q_n^m, q_{mon})$, where $stable(q^m) = (q_1, \ldots, q_n)$. We have two cases:

– If $is\_stable(q^m)$, then $q^m \xrightarrow{a}_m r^m$ and $(r^m, r) \in R$. In this case, we can conduct the same reasoning followed for the case (ii), and consider two cases for r^m.
– If $\neg is\_stable(q^m)$, let q'^m be the next state after q'^m ($q^m \xrightarrow{\beta}_m q'^m$). Lemma **??** ensures that q'^m is stable ($is\_stable(q'^m)$). Hence, $q'^m = (q_1, \ldots, q_n, q'_{mon})$, that is $q'^m \xrightarrow{a}_m r^m$ and $(r^m, r) \in R$. In this case, we can conduct the same reasoning followed for the case (ii), and consider two cases for r^m.

A.4 Correctness of our Approach

The correctness of our approach is supported by two arguments.

First, the instrumented system is observationally equivalent to the non-instrumented system where the actions used to monitor the system are considered unobservable (Proposition **??**). It is a standard assumption in runtime verification frameworks for monolithic programs to assume that the instrumentation code does not take part in the semantics of the monitored program. Thus the behavior of a monitored monolithic program that is considered to be relevant is built by considering the original actions (present before instrumentation) to be observable, and, the behavior generated by the instrumentation code plus the code of the monitor to be unobservable. Our instrumentation thus ensures that if the initial system produces an execution, then the same execution will be produced in the instrumented system, up to the actions needed to monitor the system.

The second argument is the correctness of the verdicts produced by the monitor. This is ensured by the freshness of the data received by the monitor, and, the fact that the monitor always receives the necessary information. Indeed, if the state of the system is modified in such a way that it influences the truth-value of the monitored property, it means that at least one atomic proposition of one event in the specification has possibly changed. Then, according to the definition of the function c_v, the new values of the involved elements in the specification are transmitted to the monitor. Lemma **??** and the priorities given to the interactions of the monitor ensures that the system cannot move before the monitor has finished to treat the new state and has produced a verdict.